AEROSPACE REPORT NO.
ATR-92(2778)-12

SDVS 11 Tutorial

30 September 1992

Prepared by

T. K. MENAS
Computer Systsems Division

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

Aerospace Report No.
ATR-92(2778)-12

SDVS 11 TUTORIAL

Prepared by

T. K. Menas
Computer Systems Division

30 September 1993

Engineering and Technology Group
THE AEROSPACE CORPORATION
El Segundo, CA 90245-4691

Prepared for

NATIONAL SECURITY AGENCY
Ft. George G. Meade, MD 20755-6000

PUBLIC RELEASE IS AUTHORIZED

Report No.
ATR-92(2778)-12

SDVS 11 TUTORIAL

Prepared

T. K. Menas

B. H. Levy, Principal Investigator
Computer Assurance Section

Approved

D. B. Baker, Director
Trusted Computer Systems Department

C. A. Sunshine, Principal Director
Computer Science and Technology Subdivision

Abstract

This report is a tutorial for the State Delta Verification System (SDVS), an automated system developed at The Aerospace Corporation for use in formal computer verification. SDVS helps users write and check mathematical proofs of computer correctness at the hardware, firmware, and software levels. Currently, SDVS is capable of verifying properties of computer descriptions or programs written in three computer languages. These languages are subsets of the hardware description languages VHDL and ISPS, and of the Ada programming language. In addition, SDVS may be used to verify the validity of a large class of formulas of first-order temporal logic. This tutorial contains a description of most, but not all, of the proof capabilities of SDVS. (The *SDVS 11 Users' Manual* [1] should be consulted for a more comprehensive account.) The tutorial description is embedded in numerous examples of proofs in SDVS.

Contents

1 Introduction

The purpose of this tutorial is to introduce the reader, via examples, to the State Delta Verification System (SDVS), an automated system developed at The Aerospace Corporation for use in formal computer verification.[1]

SDVS is a prototype[2] of a production-quality verification system that may be used to formally verify software from the microcode level to high-level applications programs, and hardware from the gate-level to high-level architecture. This prototype is based on a formal theoretical framework [2] and has a practical, interactive system for constructing mathematical proofs [1]. Currently, the software level of SDVS supports Ada [3] programs, the microcode level supports either ISPS [4] or VHDL [5] hardware descriptions, and the hardware level supports VHDL hardware descriptions.

In Section 2 of this tutorial, we present a brief overview of SDVS and its temporal logic. We define the central concept of SDVS, the state delta, and provide several examples that illustrate its syntax and semantics.

Sections 3 and 6 are the heart of the tutorial. The former is devoted to the most important dynamic proof commands of SDVS and the latter to the static proof commands.

The SDVS integer, bitstring, and array data types are discussed in Section 4. Other types are presented in Section 7.2.

In Section 5 we discuss the use of quantification in SDVS and prove a state delta that contains existential and universal quantification over array indices.

In Section 7 we build on the previous sections and present proofs of correctness of Ada, VHDL, and ISPS programs.

[1] For a more detailed account of SDVS, the reader should refer to the *SDVS 11 User's Manual* [1].

[2] The work on SDVS is ongoing.

2 An Overview of SDVS

The formal framework of SDVS relies on the language and techniques of mathematical logic. SDVS is based on a specialized temporal logic whose characteristic formulas, called *state deltas*, provide an operational semantic representation of computation. Our operational model is discussed in more detail in the next section. Technically, SDVS checks proofs of state deltas. SDVS can handle proofs of claims of the form "if P is true now, then Q will become true in the future." Assuming P represents a program (perhaps with some initial assertions) and Q is an output assertion, this is an input-output assertion about P. SDVS can be used as well to prove a claim of the form "if P is true now, then Q is true now;" assuming both P and Q represent programs, this claim asserts the *implementation correctness* of P with respect to Q [6]. This is a claim that one program correctly implements another. Specifications of programs may be directly formulated in state deltas, or may be programs that can be translated into state deltas.

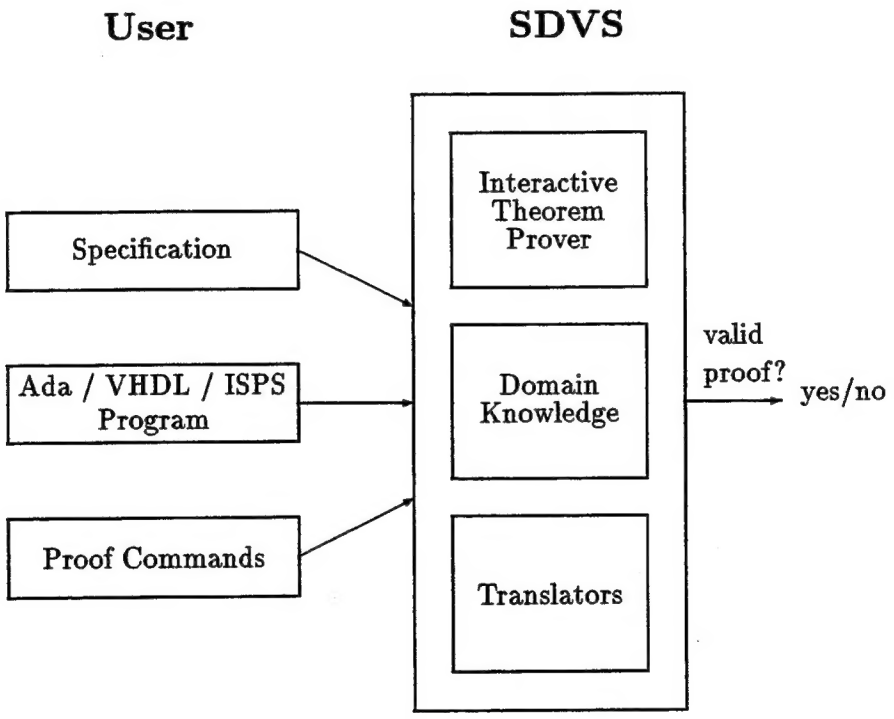

Figure 1: SDVS User Interaction

Figure 1 gives a high-level view of how a user typically interacts with SDVS.

SDVS has a theorem prover (also referred to as a proof checker), knowledge about several computer domains (data types), and a set of translators. A user inputs either an Ada, VHDL, or ISPS program together with a specification for that program. Then the user interacts with SDVS to construct a proof that the program satisfies the specification. A proof may be developed interactively and then later executed in batch mode.

Figure 2: A Timeline

The user communicates with SDVS through several languages. The *user interface language* is used for interactive proof construction. The *proof language* is used to write a proof for the system to check. The *state delta language* is used to express claims to be proven and to describe the relevant programs and specifications. Finally, the *application languages* (currently, subsets of ISPS, VHDL, and Ada) are used to express the computational objects to be verified. The *translators* function as SDVS's interface to application languages by translating them into the state delta language; the translator for each application language is an implementation of a denotational semantics for the language in terms of state deltas.

SDVS has knowledge about domains used in the programs. A main component of the theorem prover is the SDVS *Simplifier*, which implements these domains as *theories* with complete or partial *decision procedures* (or *solvers*) [7]. The decision procedures are used to deduce properties about domain objects. The complete decision procedures automatically answer queries about propositions, equality, enumeration orderings, fragments of naive set theory, and part of integer arithmetic. The partial decision procedures are part automatic and part manual, with the user instructing the system to use various axioms to deduce properties. The *domain axiomatization* is "hardwired" in SDVS, although we are currently experimenting with a facility for user-defined domains [8] that is based on the Boyer-Moore theorem prover [9]. Domains for which there are partial solvers include integer arithmetic, bitstrings, arrays, VHDL time, and VHDL waveforms. The Simplifier handles combinations of theories according to the Nelson-Oppen algorithm for cooperating decision procedures [10].

2.1 The Operational Nature of SDVS

SDVS provides an operational approach to formal verification. Operational verification systems equate a program with the class of all possible computation sequences[3] (executions) of that program; a verification system is used to show that a program is correct for *all* possible computation sequences. In SDVS a computation sequence is a model of a formula in the language of a temporal logic. Thus, correctness properties of programs can be expressed and proved in a temporal logic framework; a proof of program correctness is a mathematical proof of a temporal formula.

Every program written in a computer language accepted by SDVS describes a class of temporal structures that are possible executions of the program. For a program P with program variables (or registers) x, y, and z, and for a set of initial values (fixed or symbolic) for these variables, a possible execution of P generates a linear sequence of times t_0, .., t_i,

[3] A computational sequence of a program is a temporal structure that is a model of the program. Temporal structures and models are defined in Section 2.2.2.

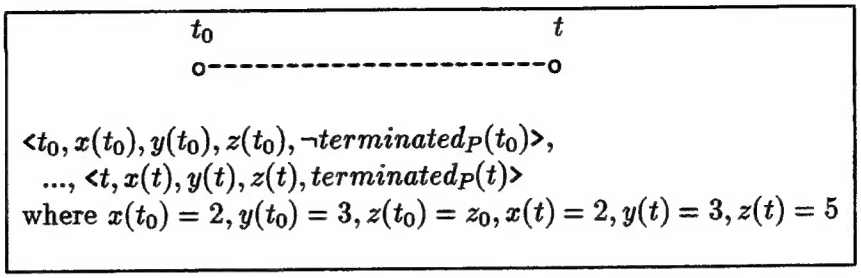

Figure 3: A Model of P

..., t_j, ..., where t_0 is the initial time. We call this sequence a *timeline*; it is illustrated in Figure 2. In our terminology, time is an abstraction that indexes the states of an execution; it is not to be confused with the ticks of a clock. At each time t, the program variables have fixed or symbolic values. For a variable x, let $x(t)$ denote its value at time t. A *state* is an ordered set consisting of a point of time t in the timeline of the execution of the program, followed by the values of the program variables at time t. For example, $<t, x(t), y(t), z(t)>$ represents the state at time t for an execution of the program P. A *model* of P is a sequence of states representing a possible execution of P.

Assertions about the program P are, in effect, assertions about the models of P. For example, if program P calculates the sum of x and y and stores the result in z, then a correctness assertion about P is that "for every pair of initial values of x and y, P terminates and, upon termination, the value of z is the sum of the initial values of x and y." This may be stated by the formula q_1:

$$q_1 \equiv \exists t(z(t) = x(t_0) + y(t_0) \wedge terminated_P(t))$$

where $x(t_0)$ and $y(t_0)$ are symbolic values, and where $terminated_P$ is **false** until P terminates, at which time it becomes **true**. The formula q_1 is true in every model M of P. Figure 3 shows an example of a model of P.

Just as assertions about a program P are assertions about all the models of P, proofs about P are proofs about all the models of P. In SDVS, when one writes a correctness proof about a program, the program is first translated into SDVS's temporal language, and then the proof is performed in that language, using the logic of SDVS. We use $tr(P)$ to denote such a translation of a program P. The translation process is akin to compilation, in that the program is "compiled" into a temporal logic formula.

For our example program P with its specification q_1, a proof that P is correct with respect to q_1 is a proof that the formula $tr(P) \rightarrow q_2$ (where q_2 is a translation of q_1 into a temporal formula) is a valid formula of temporal logic and hence is true in all temporal structures.

2.2 The State Delta Language

Statements involving time can be expressed in temporal languages and proved in temporal logics. Temporal languages have symbols, called *temporal operators*, that are used to

express such statements. The only temporal operator of SDVS is the **state delta.**[4] It is a combination of the classical temporal operators **always**, **eventually**, and fragments of **until**. In this section we briefly discuss its syntax and semantics.

2.2.1 Syntax

The language L of SDVS contains function, predicate, and constant symbols and two types of variables, global and local.[5] The values of the global variables are constant throughout a timeline (computation), whereas the values of the local variables vary with time. The atomic terms are either constants (e.g. 0 and 1), global variables, or of the form $.x$ or $\#x$, where x ranges over the local variables.

In a manner analogous to that for the predicate calculus, terms and atomic formulas are defined from the atomic terms, the function symbols (e.g. $+$ and $*$), and the predicate symbols (e.g. *gt* and *le*). For example, $2*(\#x-.y+a)$ is a term with local variables x and y and global variable a, and $2*(\#x-.y+a)=.y$ is an atomic formula.

The set of formulas of SDVS is defined to be the smallest set that contains the atomic formulas and that is closed under conjunction ("and"), disjunction ("or"), negation ("not"), implication ("implies"), universal quantification over global variables ("forall a"), existential quantification over global variables ("exists a"), and the state delta operator

```
[sd  pre: p  comod: c  mod: m  post: q]
```

where (1) the precondition p is a formula with the property that for every local variable x, every occurrence of $\#x$ in p is an occurrence in either the precondition or postcondition of a state delta subformula of p; (2) the postcondition q is a formula; and (3) the comodification list c and modification list m are lists of local variables.

For every local variable x and formula ϕ, an occurrence of $.x$ ($\#x$) in ϕ is an *upper level occurrence* if and only if the occurrence is not in the precondition or postcondition of a state delta subformula of ϕ. A formula ϕ is of *precondition type* iff for every local variable x, ϕ has no upper-level occurrences of $\#x$. A precondition type formula is *static* (nontemporal) if and only if it has no state delta subformulas. Here are some examples:

(i) The first occurrence of $\#x$ in the formula

```
#x le 2*.y and [sd  pre: true  comod: x,y  mod: x  post: #x=.x+1]
```

is an upper-level occurrence, whereas the second occurrence is not. This formula is not a precondition type formula.

(ii) The formula

[4] State deltas were first introduced in [11].

[5] Global and local variables may be constrained to range over specific SDVS data types (see Section 4). By default, they are usually of type integer.

```
.x le 2*.y implies [sd pre: .x lt .y  comod: x  mod: z  post: #x=1]
```

is of precondition type but is not static.

(iii) The formula

```
exists a (.x+.y = 5 and a ge 3)
```

is a static formula.

2.2.2 Semantics

A temporal structure M consists of a first-order base structure (e.g. the integers), a timeline T, and a valuation V, such that

(i) V assigns a function, predicate, and element of the base structure to every function, predicate, and constant symbol of the language, respectively.

(ii) To every global variable a, V assigns an element, $V(a)$, of the base structure.

(iii) For every local variable x and every time t of the timeline T, V assigns an element of the base structure, $V(.x,t)$ [or simply $x(t)$], to the atomic term $.x$ at time t.

Let t be an element of T. Then for every term τ in which # does not occur, V assigns an element of the base structure, $V(\tau,t)$, to the term τ at time t. Furthermore, for every static precondition formula ϕ, V assigns a truth value to ϕ, $V(\phi,t)$, at time t, in a manner analogous to that for the predicate calculus.

For example, if $V(a) = 2$, $V(.x,t) = -1$, and $V(.y,t) = 4$, then

$$V((.y = 3 * a + 2 * .x), t)) = true$$

and

$$V(\exists\ b\ \ (b = a\ \ and\ \ .y = 3 * b + 2 * .x), t) = true$$

We proceed to define $V(\phi,t)$ for every precondition formula ϕ and every t in T. The definition is by induction on the complexity of ϕ.

Boolean operators and quantification over global variables are treated in the standard way.

Suppose that ϕ is the state delta formula

```
[sd  pre: p  comod: c  mod: m  post: q]
```

and that $.x_1,\ldots,.x_n$ and $\#y_1,\ldots\#y_k$ are the upper-level (local) atomic terms of q. Then ϕ is true at time t if and only if for every $t_1 \geq t$ such that every local variable in c is constant in the closed time interval $[t,t_1]$ and such that p is true at time t_1, then there is a

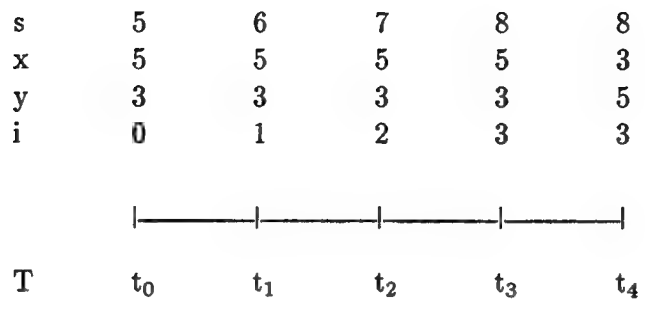

Figure 4: A Temporal Structure M

time $t_2 \geq t_1$ such that every local variable not in m is constant in the closed time interval $[t_1, t_2]$ and such that

$$q[V(.x_1, t_1)/.x_1, \ldots, V(.x_n, t_1)/.x_n \ ; \ .y_1/\#y_1, \ldots, .y_k/\#y_k]$$

is true at time t_2. The substitution of the value of the x_i's at time t_1 for the $.x_i$'s is made prior to the substitution of the $.y_j$'s for the $\#y_j$'s in q. In effect, the upper-level atomic terms of q are evaluated at time t_1 if they are of the form $.x$, and at time t_2 if they are of the form $\#y$.

In the definition of the truth of the state delta ϕ, a time with the properties of t_1 described above is said to be a "precondition time" of the state delta (with respect to the "current" time t at which it is evaluated), and a time with the properties of t_2 is said to be a "postcondition time" of the state delta (with respect to t_1). A modification or comodification list of "all" is an abbreviation for the list (set) of all local variables of the language. (modification) list of a state delta is (modification) field delta. If the comodification list of the state delta ϕ is "all," then every precondition time is, in effect, the same as the current time,[6] i.e., $t_1 = t$. Similarly, if the modification list of the state delta is empty, then every postcondition time of the state delta is, in effect, equal to its corresponding precondition time, i.e. $t_1 = t_2$.

Evidently, the state delta operator is complex. A few examples should clarify its semantics. Suppose that x, y, s, and i are the only local variables of the language, a and b are global variables, and the base first-order structure is the set of integers with functions "+", "-", "*" and predicates "le", and "lt".

Figure 4 depicts a temporal structure M with five points in its timeline. The base structure is the set of integers with the usual functions and predicates. The numbers in each column are the values of the local variables at each point of the timeline.

The discussion that follows refers to the structure M and to its timeline.

[6] An important fact about every precondition formula σ of SDVS is that for every temporal structure, if $t \leq t_1$ are elements of the timeline of the structure such that every local variable of the language is constant in the time interval $[t, t_1]$, then the truth value of σ is constant in $[t, t_1]$.

(i) The static formula $.i + .y + .x = .s + 3$ is true at t_1.

(ii) A state delta with comodification and modification lists of "all" and precondition "true", asserts that there is a time in the future (possibly now) such that the post-condition is true. For example, the state delta

```
[sd  pre: true  comod: all  mod: all  post: #i=1]
```

is true at t_0 and t_1 and false at all other times.

(iii) The state delta

```
[sd pre: true comod: all mod: s,i post: #s=.s+1 and #i=.i+1]
```

is true at times t_0, t_1, and t_2, but false at all other times. The comodification list "all" denotes the list of all local variables. At any particular time r in the timeline of a temporal structure, this state delta is true iff there is a time $t \geq r$ such that only s and i may change their value in the closed time interval $[r, t]$ and $s(t) = s(r) + 1 \wedge i(t) = i(r) + 1$.

(iv) The state delta

```
[sd   pre: true  comod: all  mod: x,y  post: #y=.x and #x=.y]
```

is true at t_3 but false at all other times.

(v) A state delta with a comodification list of "all" and an empty modification list asserts that the precondition implies the postcondition at the current time. The reason for this is that any precondition time is, in effect, equal to the current time, and any postcondition time is, in effect, equal to the corresponding precondition time. Thus the state delta

```
[sd pre: .i=0  comod: all  mod:   post:  #s=5]
```

is true at time t_0. In fact, it is true at all other times as well, because its precondition is false at every other time.

(vi) A state delta with empty comodification and modification lists asserts that at every time t in the future, the precondition at time t implies the postcondition at time t. For example, the state delta

```
[sd pre: true  comod:   mod:   post: #i ge 1]
```

is true at time t_1, since the value of i is greater than or equal to 1 at t_1 and thereafter. However, it is false at t_0.

(vii) The state delta

```
[sd pre: .i lt .y
  comod: x,y
    mod: s,i
   post: #s=.s+1 and #i=.i+1]
```

is true at t_0. The precondition times with respect to the current time t_0, are t_0, t_1, and t_2; for these precondition times the corresponding postcondition times are t_1, t_2, and t_3, respectively. Note that t_3 and t_4 are not precondition times with respect to t_0, because the precondition is false at the former time, and the comodification list is violated in the latter time (y is not constant in the interval $[t_0, t_4]$). This state delta is also true at t_1 and t_2. It is true at t_3 by default, because of its comodification list and the fact that its precondition is false at t_3, and it is false at t_4.

(viii) The "nested" state delta

```
[sd pre: true
  comod: all
    mod: i,s
   post: #i=.i+1 and #s=.s+1 and
         [sd pre: true
           comod: all
             mod: i,s
            post: #i=.i+1 and #s=.s+1]]
```

is true at t_0 and t_1, but false at all other times. It is not true at t_2 because the state delta in its postcondition is false at t_3.

A precondition formula is *valid with respect to the first-order structure A* iff it is true at the initial point of every temporal structure M whose first-order base structure is A. For example, if the base structure A is the set of integers, then the state deltas

```
[sd  pre: .x=1 and
          [sd  pre: true  comod: all  mod: y  post: #y =.x+5]
   comod: all
     mod: y
    post: #y=6]
```

and

```
[sd  pre: .x=a and .y=b and
          [sd  pre: true
             comod: all
               mod: x,y
              post: #x=.y and #y=.x]
   comod: all
     mod: x,y
    post: #x=b and #y=a]
```

are valid with respect to A.

In SDVS the only formulas that may be proved (valid) are the state deltas. But this is not an important limitation of the system, because for any precondition formula S, the state delta

```
[sd  pre: true  comod: all  mod:  post:  S]
```

is valid iff S is valid.

2.3 Model of Storage

Although we have used the local variables (places in SDVS terminology) as if they were independent of each other (so that, for example, a change of one does not affect the other), places were historically considered to be memory locations that could possibly overlap. Thus in situations in which they are considered to be independent, SDVS must be explicitly informed of that fact. Thus, for the above examples it would be necessary to add the statement "covering(all,x,y,i,s)" in the precondition of every upper-level state delta. This statement asserts that the local variables x, y, i, and s are independent of each other and that they comprise the set of all local variables. The discussion of ISPS (Section 7.3) will describe the possible overlap of places in greater detail.

2.4 Proofs in SDVS

A proof is a structured argument, using mathematical logic, that a formula is true. The state delta language is used to write theorems (formulas) to be proved. Using the proof language in SDVS, the user has access to axioms and rules with which to write interactively a proof that the system checks. If a state delta is proved in SDVS, then it is true in all temporal structures (computational models) with the appropriate base structure.

The underlying proof method used by SDVS is *symbolic execution*. Symbolic execution essentially involves *executing* a program or machine description from its initial state through successive states, using *symbolic* values for the program variables or for the contents of machine registers. Of course, the computation path is often conditional on specific values; in these instances subproofs must be initiated to account for all possibilities. The correctness claims that are proved are all of the form "At certain states some conditions are true." Thus, during a proof there are two kinds of tasks: to go from state to state, and to prove that certain things are true in a given state. These are the dynamic and static aspects of the proof system, respectively.

The dynamic proof language has three basic rules: straight-line symbolic execution (for instances where the path is not data dependent), proof by cases (at branch points), and induction (necessary when the number of times through a loop is data dependent, but could also be used for a large constant number of iterations). There are other variations to handle special cases, such as a command to handle general recursive procedures.

Once SDVS has "arrived" at a state that the user knows (or hopes) will satisfy the conditions to be proved, SDVS must be convinced that these conditions are true. Thus, SDVS has some explicit facts about the state listed in its database, which perhaps do not include verbatim the required condition. The problem is then to prove the "static" theorem that those facts imply the required condition. This is a theorem of ordinary mathematics. The domains associated with these theorems frequently involve bitstrings, integers, arrays, and the like. Also, a knowledge of basic propositional logic, equality, and some quantification theory is often needed.

In these domains (and others) SDVS has a mix of automatic deduction capability and axioms that may be invoked by the user when proving theorems. As mentioned above, there are two reasons for such a mix: For theoretical reasons of impossibility or inefficiency, some deductions cannot be done automatically, or else a totally nonautomatic deduction capability would be too time-consuming for the user.

2.5 Installing SDVS

SDVS is available on magnetic tape in four different formats: source code; object code for Franz Allegro Common Lisp (FACL); object code for Lucid Common Lisp (LCL); and as a standalone executable utilizing the Franz Allegro Runtime package. Each format requires its own procedure for creating or loading SDVS, as outlined below. However, the procedure for reading the system files from the tape is the same for all formats.

SOFTWARE REQUIREMENTS

SDVS currently runs under Franz Allegro Common Lisp release 4.1 and Lucid Common Lisp 4.1. SDVS is also available as a standalone executable utilizing the Franz Allegro Runtime package; users of this version of SDVS are not required to supply their own Common Lisp environment.

DISK SPACE REQUIREMENTS

Table 1 gives the disk space requirements for SDVS 11. "Installed" represents the disk requirements of the system after SDVS has been installed, and assumes that the tar archive has been recompressed. The size of your installed executable image, if you are building SDVS from the source or either binary version, will depend on the size of your (vanilla) Common Lisp image. These numbers are therefore approximate. All numbers are in megabytes (MB).

READING THE SYSTEM FILES

First, you should create a top-level directory to contain all of the files and subdirectories associated with SDVS. In our system, this directory is called *versys* (for VERification SYStem) and resides as a subdirectory under */u* giving */u/versys*. Although you can give your directory any name, we suggest you use the same name for compatibility; yours can be located anywhere, however. For example, you might put it as a subdirectory of */usr/lib*, giving */usr/lib/versys*. For the examples below, we assume you have */usr/lib/versys* as your top-level directory.

Table 1: Disk Space Requirements for SDVS 11

	To Load From Tape	Installed
Source (.lisp)	2.4	N/A
Lucid Object (.sbin)	2.7	31.5
Franz Object (.fasl)	3.7	48.1
Franz Runtime	8.9	38.9

Next, you will want to load the SDVS system tar file from the tape. To do this, create a *tmp* directory in your top-level *versys* directory, connect (*cd*) to it, and extract (*tar*) the system tar file as follows ([unix] is the system prompt):

[unix] *tar xfmv xxx*

where *xxx* is the device name for your tape drive, e.g. */dev/rst0*. This will create a file named `sdvs`*nn*`-`*xxxx*`.tar.Z` where *nn* is the current release number (e.g. `11`) and *xxxx* is `lisp` (for source files), `sbin` (for LCL object), `fasl` (for FACL object), or `runtime` for FACL Runtime. The file is compressed, so it must be uncompressed:

[unix] *uncompress sdvsnn-xxxx.tar*

replacing *nn* and *xxxx* appropriately.

Now, the system directories must be extracted from the tar file:

[unix] *tar xfmv sdvsnn-xxxx.tar*

This process creates a file structure containing the individual files from which the SDVS system can be used or built. Once this process is complete, you may delete `sdvs`*nn*`-`*xxxx*`.tar` if you feel you have no further need for it. An alternative is to recompress the file. Both will save disk space.

[unix] *compress sdvsnn-xxxx.tar*

Before you can build and use an SDVS executable image or use the FACL Runtime executable, you must define a UNIX environment variable as follows. This can be done directly in the shell in which you plan to build or use SDVS or by adding the command to your `.cshrc` file.

[unix] *setenv SDVS_DIR "/usr/lib/versys/"*

Of course, you will need to supply the correct path you have chosen for your top-level directory. Please note the slash (/) character at the end; it is required.

BUILDING AN SDVS EXECUTABLE IMAGE

Once you have all of the system files available, you can build an executable SDVS image. To do this, you must start up a (vanilla) Common Lisp session (either LCL or FACL) and load the *init-sdvs.lisp* file found in your top-level directory. (If you don't know how to start

up a Common Lisp session, see your system administrator.) For example, to load the file, type

> *(load "/usr/lib/versys/init-sdvs")*

After the *init-sdvs.lisp* file has been loaded, you are ready to tell Lisp to build your SDVS executable. Two functions will do this: *make-sdvs* builds from the object files; *make-new-sdvs* builds from the source files and compiles the entire system. Each function takes one argument, the name you wish to give the executable; the executable will automatically reside in your top-level directory. You may give the executable any name you want; in the following examples, we use the name *sdvs11* for our executable. Each of these functions will produce a trace of what is happening. (NOTE: For these operations, you must have write privileges to the appropriate directories.)

For creating an SDVS executable from source:

> *(make-new-sdvs "sdvs11")*

For creating an SDVS executable from binary:

> *(make-sdvs "sdvs11")*

You may safely ignore any warning message printed by the system. When you return to the Lisp prompt, you can exit Lisp by

> *(quit)*

USING THE SDVS RUNTIME EXECUTABLE

If you have extracted the SDVS system files from a tape containing the "runtime" format, the file `/usr/lib/versys/sdvs11` (assuming the appropriate top-level directory) contains the executable image. This can be used to run SDVS directly, as noted below.

RUNNING SDVS

You have gone through this procedure and have created your executable. How do you run SDVS? At the Unix shell, just type, for example

`[unix]` */usr/lib/versys/sdvs11*

or just *sdvs11* if you are connected (*cd*) to the top-level directory (*/usr/lib/versys* in our example) or if your $PATH environment variable contains the path to the top-level directory.

RUNNING THE TEST SUITE

Included in the SDVS release is a set of tests that exercise the system. To run these tests, you must first start up SDVS. (After building your SDVS executable, you should restart SDVS so that the system is initialized properly.) When you get to the SDVS prompt, you will want to *eval*uate the *expression*, invoking the tests as follows:

```
<sdvs.1>  eval
   expression:    (run-sdvs-test-proofs)
```

A very long trace will appear. If the tests run successfully (this may take over two hours on a Sun 4), you will return to the SDVS prompt. If something goes wrong, Lisp will "break," allowing you to examine the system; Lisp will print out some diagnostic information and put you at a prompt. If this should happen, you may exit Lisp by typing *(quit)*.

You may restart SDVS by first returning to the top level of Lisp and invoking the function *sdvs* as follows:

> (*sdvs*)

From the SDVS prompt, you can return to Lisp by typing the SDVS command **bye**.

3 Dynamic Execution

In this section we present most of the SDVS commands that advance the state of a computation or program execution. (Section 6 is devoted to those commands that do not advance the state, but rather enlarge the set of facts known to SDVS about a specific state.) In Section 3.1 we consider those commands that are most often used in proofs that involve the translation of assignment statements, in Section 3.2 those commands that involve the translation of "case" program segments, and in Section 3.4 the **induct** command that is used in proofs that involve "loop" program segments. Finally, in Section 3.3 we consider ways of proving state deltas that are assertions about the current state or about all future states.

Henceforth, in the system-user dialogue, `typewriter print` is system output and *italic print* is user input. In the discussion of the examples, mathematical formulas and terms are printed in TEX math mode.

3.1 Straight-line Proofs

In this section we present in a very leisurely fashion two simple examples that will introduce the reader to an extensive part of the SDVS proof environment.

Example 1 In the first example we prove that the state delta translation of the program P in Section 2.1 implies its specification, namely, that if the initial values of x and y are 2 and 3, respectively, and z is assigned the value of $x + y$, then there will be a time when the values of x, y, and z will be 2, 3, and 5, respectively. Lest the reader be alarmed, we note that the assignment of concrete values to the variables is only for pedagogic reasons: in most of our examples, the values of the program variables will be symbolic.

We first create the state delta that corresponds to the assignment statement

```
z := x + y
```

using the **createsd** command.

```
<sdvs.1>  createsd
      name:  assign.sd
   [SD pre:  true
   comod[]:  all
     mod[]:  z
      post:  #z=.x+.y
   ]
```

If *assign.sd* is true at time t_0, then there is a time $t_1 \geq t_0$ such that $z(t_1) = x(t_0) + y(t_0)$ and such that the values of x and y remain constant in the interval $[t_0, t_1]$ (because of the

modification list, only z may change its value in this time interval). Hence $x(t_1) = x(t_0)$, and $y(t_1) = y(t_0)$.

The pretty-print command **pp** displays the state delta associated with a given state delta name:

```
<sdvs.1>  pp
   object:  sd
   state delta name:  assign.sd

 [sd pre: (true)
   comod: (all)
     mod: (z)
    post: (#z = .x + .y)]
```

We now create the state delta that asserts that the state delta translation of P implies the specification of P.

```
<sdvs.1>  createsd
      name:  example1.sd
   [SD pre:  covering(all,x,y,z), .x=2, .y=3, formula(assign.sd)
   comod[]:  all
     mod[]:  z
      post:  #x=2, #y=3, #z=5
   ]
```

Note that the proper way to include the state delta *assign.sd* in the precondition (or postcondition) of *example1.sd* is to include it with "*formula*(*assign.sd*)". Also note that commas at the top level of the precondition and postcondition of a state delta are interpreted as "and".

The initialization command **init** should always be used prior to the beginning of a top-level proof. The command clears any knowledge that the system has acquired in a given session (apart from the already established association of names with formulas).

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.
```

SDVS has a list of flags that may be set by the user:

```
<sdvs.1>  flags
```

```
abbreviationlevel       = none
acceptfileproofs        = on
autoclose               = on
checkexistence          = off
checksyntax             = on
displaympsds            = on
ekltraceflag            = off
enumerate               = off
invariance              = off
optimizeassignments     = simp
ppdottednames           = off
pplinewidth             = 75
reportpropagations      = on
showstats               = off
showstep#               = off
strongcoverings         = off
stronglytyped           = off
traceflag               = on
uniquenamelevel         = 1
weaknext_tr             = off

Type 'help flags' for a description.
```

If the autoclose flag is set to "off", SDVS will not usually "close"[7] the proof of a state delta, even if the goal (the postcondition) of the state delta has been achieved (reached).

```
<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[off]:  off

 setflag autoclose -- off
```

We are now ready to prove *example1.sd*.

```
<sdvs.2>  prove
   state delta[]:  example1.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x,y,z),.x = 2,.y = 3,formula(assign.sd))
          comod: (all)
            mod: (z)
           post: (#x = 2,#y = 3,#z = 5)]
```

[7] A proof of a state delta is closed if the proof is complete.

```
Complete the proof.
```

SDVS has now opened the proof of *example1.sd*: it has advanced the state (subject to the constraints of the comodification list "all") to a time at which the precondition of the state delta is asserted (to be true), and has placed the translation of the postcondition at the top of its goal stack. It has also noted the modification list "z"; by doing so, any advancement of the state must henceforth be made subject to this modification list, that is, any advancement must be restricted to possible changes in the value of z only.

Since the precondition has been asserted to be true, the values of x and y must be 2 and 3, respectively. This may be checked by the **simp** (simplify) command. Recall that the current value of a place a is denoted by $.a$ and not by $\#a$.

```
<sdvs.2.1>  simp
   expression:  .x

 2

<sdvs.2.1>  simp
   expression:  .y

 3
```

The value of z at this point is symbolic and indeterminate:

```
<sdvs.2.1>  simp
   expression:  .z

 z\5
```

Since *assign.sd* is in the precondition, it is now true. This may be ascertained by the **usable** query which displays the state deltas (and the quantified formulas) that are true in the current state.

```
<sdvs.2.1>  usable

 u(1) [sd pre: (true)
        comod: (all)
          mod: (z)
         post: (#z = .x + .y)]


 No usable quantified formulas.
```

A state delta that is true at the current time may not be "applicable." To be applicable, its precondition must also be true at the current time. Since the precondition of *assign.sd* is always true, *assign.sd* is now applicable. This may be checked by the **nsd** command, which displays the most recent state delta that the system knows to be applicable.

```
<sdvs.2.1>  nsd

 [sd pre: (true)
   comod: (all)
     mod: (z)
    post: (#z = .x + .y)]
```

At any point in the course of a proof the user may ask SDVS to list the goals of the most current proof that it does not know to be true:

```
<sdvs.2.1>  whynotgoal
   simplify?[no]:  <CR>

 g(3) #z = 5
```

The **apply** command is used to advance the state by "applying" an applicable state delta whose modification list is a sublist of the modification list of the state delta to be proved. If no argument is given, SDVS applies the most recent applicable state delta:

```
<sdvs.2.1>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (z)
              post: (#z = .x + .y)]
```

SDVS executes the application of a state delta S by

- linking every upper-level dotted place a in the postcondition of S to any information about a it currently has,
- removing any information about the current state whose truth depends on the values of places in the modification list of S, and
- asserting the postcondition of S.

Thus the application of *assign.sd* advances the state to a time at which x and y have retained their previous values and at which the value of z is asserted to be equal to the sum of these two previous values. Furthermore, *assign.sd* is no longer known to be true in

this state (because its comodification list did not allow anything to change) and is thus no longer usable.

Let us check these facts:

```
<sdvs.2.2> simp
   expression: .x

 2


<sdvs.2.2> simp
   expression: .y

 3


<sdvs.2.2> simp
   expression: .z

 5


<sdvs.2.2> usable

 No usable state deltas.


 No usable quantified formulas.
```

Thus our goal has been reached. SDVS has not automatically closed the proof because the "autoclose" flag is off.

Let us check the goals once more:

```
<sdvs.2.2> whynotgoal
   simplify?[no]: <CR>

 The goal is TRUE.  Type 'close'.
```

The command **close** will close the proof:

```
<sdvs.2.2> close

 close -- 1 steps/applications
```

Once a proof of a state delta is closed, the state delta becomes true (usable) but any information gained during its proof is lost and any information that was lost after its proof was opened is restored: the state is "popped" to the time before the "prove" command was used to prove it:

```
<sdvs.3>  simp
   expression:  .x

 x\7

<sdvs.3>  simp
   expression:  .y

 y\8

<sdvs.3>  simp
   expression:  .z

 z\9

<sdvs.3>  usable

 u(1) [sd pre: (covering(all,x,y,z),.x = 2,.y = 3,formula(assign.sd))
        comod: (all)
          mod: (z)
         post: (#x = 2,#y = 3,#z = 5)]


 No usable quantified formulas.
```

Note that although *example1.sd* is usable, it is not applicable, because its precondition is not necessarily true:

```
<sdvs.3>  nsd

 No applicable state deltas.
```

A proof that has just closed may be given a name and stored temporarily in the system (for the duration of the current session only) using the **dump-proof** command:

```
<sdvs.3>  dump-proof
   name:  example1.sd.proof

 Current proof dumped to example1.sd.proof.
```

This command must be given prior to an **init** command.

Now let us initialize the system once more to demonstrate that **init** will erase *example1.sd* from the usable list:

```
<sdvs.3>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.


<sdvs.1>  usable

 No usable state deltas.


 No usable quantified formulas.
```

The saved proof "example1.sd.proof" may be run in batch mode by means of the **init** or **interpret** commands.

```
<sdvs.1>  init
   proof name[]:  example1.sd.proof

State Delta Verification System, Version 11

Restricted to authorized users only.

 setflag autoclose -- off

 open -- [sd pre: (covering(all,x,y,z),.x = 2,.y = 3,formula(assign.sd))
          comod: (all)
            mod: (z)
           post: (#x = 2,#y = 3,#z = 5)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (z)
              post: (#z = .x + .y)]

 close -- 1 steps/applications
```

It may also be pretty-printed by the **pp** command:

```
<sdvs.3>  pp
   object:  proof
   proof name:  example1.sd.proof

 proof example1.sd.proof:

  (setflag autoclose off,
   prove example1.sd
     proof:
       (apply u(1),
        close))
```

But this is true only during the current session. To store in a file the proof and the state deltas created in this SDVS session, the user may write them by means of the **write** command:

```
<sdvs.3>  write
   path name[testproofs/foo.proofs]:  tutorial/example1
                state delta names[]:  assign.sd, example1.sd
                      proof names[]:  example1.sd.proof
                      axiom names[]:  <CR>
                      lemma names[]:  <CR>
                    formula names[]:  <CR>
                   formulas names[]:  <CR>
                      macro names[]:  <CR>
                   datatype names[]:  <CR>
                   adalemma names[]:  <CR>


 Write to file "tutorial/example1" -- (assign.sd,example1.sd,
                                       example1.sd.proof)
```

Furthermore the association of names with formulas in the current session may be severed by the **delete** command:

```
<sdvs.3>  delete
   object type:  proof
   object name:  example1.sd.proof
```

```
<sdvs.3>  delete
   object type:  sd
   object name:  assign.sd

<sdvs.3>  pp
   object:  sd
   state delta name:  assign.sd

The name assign.sd is not associated with a state delta.

 pp error: unknown state delta

<sdvs.3>  pp
   object:  proof
   proof name:  example1.sd.proof

The name example1.sd.proof is not associated with a proof.

 pp error: unknown proof
```

In any new session with SDVS, these associations may be read from the file

```
<sdvs.3>  read
   path name[tutorial/example1]:  tutorial/example1

 Definitions read from file "tutorial/example1"
   -- (assign.sd,example1.sd,example1.sd.proof)
```

and executed via the **init** command or the **interpret** command:

```
<sdvs.1>  interpret
   proof name:  example1.sd.proof

 setflag autoclose -- off

 open -- [sd pre: (covering(all,x,y,z),.x = 2,.y = 3,formula(assign.sd))
          comod: (all)
            mod: (z)
           post: (#x = 2,#y = 3,#z = 5)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (z)
              post: (#z = .x + .y)]
```

```
close -- 1 steps/applications
```

The main difference between the **init** command and the **interpret** command in running a batch proof is that the **interpret** command does not initialize the system prior to running the proof. Thus in the cases in which we want to run a batch proof of a state delta within the proof of another state delta, the **interpret** command is appropriate.

Example 2 This second simple example differs from the first primarily in that the local variables have only symbolic values. Proofs of such programs are said to be done by *symbolic execution.* The example is a proof that the state delta translation of the program segment Q

```
temp := x;
   x := y;
   y := temp;
```

implies that at the end of the execution of Q, x and y will have exchanged their initial values. The example also illustrates the translation of a program consisting of several statements to a nested state delta. The translation of the assignment statement

```
y := temp
```

is the state delta *assign.temp.to.y.sd*:

```
[sd pre: (true) comod: (all) mod: (y) post: (#y = .temp)]
```

The translation of the assignment statement

```
x := y
```

with its continuation is the state delta *assign.y.to.x.sd*:

```
[sd pre: (true)
  comod: (all)
    mod: (x)
   post: (#x = .y,formula(assign.temp.to.y.sd))]
```

The truth of *assign.y.to.x.sd* at a time t_{i_1} implies that there are times $t_{i_3} \geq t_{i_2} \geq t_{i_1}$ such that

- $x(t_{i_2}) = y(t_{i_1})$ and $y(t_{i_3}) = temp(t_{i_2})$ and

- only x may change its value in the interval $[t_{i_1}, t_{i_2}]$ and
- only y may change its value in the interval $[t_{i_2}, t_{i_3}]$.

The translation of the assignment statement

```
temp := x
```

with its continuation is the state delta *assign.x.to.temp.sd*:

```
[sd pre: (true)
  comod: (all)
    mod: (temp)
   post: (#temp = .x,formula(assign.y.to.x.sd))]
```

This state delta is in fact the state delta translation of the program segment Q. Finally, the assertion that the translation of Q implies the specification of Q is the state delta *example2.sd*:

```
[sd pre: (covering(all,x,y,temp),formula(assign.x.to.temp.sd))
  comod: (all)
    mod: (all)
   post: (#x = .y,#y = .x)]
```

Now let us open the proof of *example2.sd*:

```
<sdvs.1>  prove
   state delta[]:  example2.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x,y,temp),formula(assign.x.to.temp.sd))
           comod: (all)
             mod: (all)
            post: (#x = .y,#y = .x)]

 Complete the proof.
```

The opening of the proof of *example2.sd* asserts its precondition at the initial state of the computational model. At this state, the variables x and y are given symbolic values of the form "*variablename\number*", where "*number*" is a positive integer that is generated in an indeterminate manner. These values are listed by the **ppeq** (pretty-print equivalence class) command:

```
<sdvs.1.1>  ppeq
```

```
   expression:  .x

 eqclass = x\21

<sdvs.1.1>  ppeq
   expression:  .y

 eqclass = y\20

<sdvs.1.1>  ppeq
   expression:  .temp

 eqclass = temp\22
```

The goal of the proof of *example2.sd*, which is the interpreted postcondition of *example2.sd*, may be viewed by means of the **goals** query:

```
<sdvs.1.1>  goals

 g(1) #x = y\20
 g(2) #y = x\21
```

Since the precondition of *example2.sd* has been asserted, the state delta *assign.x.to.temp.sd* is usable (true), and moreover, since its precondition is also true at the current state, it is also applicable. This latter fact may be ascertained via the query **applicable**, to which SDVS responds with a list of all the state deltas that it knows to be applicable at the current state:

```
<sdvs.1.1>  applicable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (temp)
        post: (#temp = .x,formula(assign.y.to.x.sd))]
```

This state delta may be applied via the **apply** command with the parameters *u* and 1:

```
<sdvs.1.1>  apply
   sd/number[highest applicable/once]:  u
                              number:  1

   apply -- [sd pre: (true)
             comod: (all)
               mod: (temp)
              post: (#temp = .x,formula(assign.y.to.x.sd))]
```

The state has now been advanced to a time at which the symbolic value of *temp* is the previous value of x, and x and y have retained their values:

```
<sdvs.1.2>  ppeq
   expression:  .x

 eqclass = x\21

<sdvs.1.2>  ppeq
   expression:  .y

 eqclass = y\20

<sdvs.1.2>  ppeq
   expression:  .temp

 eqclass = x\21
```

Furthermore, at this new state, *assign.x.to.temp.sd* is no longer necessarily true, because its comodification list has a nonempty intersection with the modification list of the state delta that was applied. Thus, it is also not applicable, but *assign.y.to.x.sd* is applicable since it was asserted to be true by the application and its precondition is always true. Let us note this fact and apply the state delta by using another parameter for the **apply** command, namely, the name of the state delta to be applied:

```
<sdvs.1.2>  applicable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (x)
        post: (#x = .y,formula(assign.temp.to.y.sd))]

<sdvs.1.2>  apply
   sd/number[highest applicable/once]:  assign.y.to.x.sd

   apply -- [sd pre: (true)
             comod: (all)
               mod: (x)
              post: (#x = .y,formula(assign.temp.to.y.sd))]
```

The state has been advanced once more. Let us check the symbolic values of the variables in the computation:

```
<sdvs.1.3>  ppeq
   expression:  .x
```

```
 eqclass = y\20

<sdvs.1.3>  ppeq
   expression:  .y

 eqclass = y\20

<sdvs.1.3>  ppeq
   expression:  .temp

 eqclass = x\21
```

As expected, one of the goals has been achieved: the current value of x is the initial value of y. Once more, the goals are:

```
<sdvs.1.3>  goals

 g(1) #x = y\20
 g(2) #y = x\21
```

The query **whynotgoal** will demonstrate that one of the goals has indeed been achieved and will list the remaining ones:

```
<sdvs.1.3>  whynotgoal
   simplify?[no]:  <CR>

 g(2) #y = x\21
```

The query **ps** (proof state) will show the history of the proof:

```
<sdvs.1.3>  ps

  << initial state >>
  proof in progress of example2.sd <3>
    apply u(1) <2>
    apply assign.y.to.x.sd <1>
    --> you are here <--
```

Had we erred in the course of the proof, we would want to go back to the point before the error and start anew. This may accomplished via the **pop** command:

```
<sdvs.1.3>  pop
   number of levels[1]:  2

 2 levels popped.
```

We are now at the point before the first apply:

```
<sdvs.1.1>  ps

  << initial state >>
  proof in progress of example2.sd <1>
    --> you are here <--
```

The command **apply** has another use: with a number n as the parameter, SDVS will try to apply n state deltas, using at each point the first applicable state delta in its list:

```
<sdvs.1.1>  apply
   sd/number[highest applicable/once]:  2

   apply -- [sd pre: (true)
             comod: (all)
               mod: (temp)
              post: (#temp = .x,formula(assign.y.to.x.sd))]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (x)
              post: (#x = .y,formula(assign.temp.to.y.sd))]

<sdvs.1.3>  ps

  << initial state >>
  proof in progress of example2.sd <3>
    apply <2>
    apply <1>
    --> you are here <--
```

So we are back to the state attained after two applications.

To illustrate another important way to advance the state of computation by the application of state deltas, we pop back again

```
<sdvs.1.3>  pop
   number of levels[1]:  2

 2 levels popped.
```

and get some help on the **until** command:

```
<sdvs.1.1>  help
```

```
   with[all]: until

 until <postformula>
   Symbolically executes highest applicable state deltas until
   <postformula> is TRUE, there are no more applicable state deltas, or the
   'autoclose' flag is on and the current goal is satisfied.
```

Let us apply until the goal $\#x = .y$ has been achieved, and then check where we are in the proof:

```
<sdvs.1.1>  until
   formula:  #x=.y

   apply -- [sd pre: (true)
             comod: (all)
               mod: (temp)
              post: (#temp = .x,formula(assign.y.to.x.sd))]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (x)
              post: (#x = .y,formula(assign.temp.to.y.sd))]

   until break point reached -- #x = .y
```

One more application should close the proof, but let us first set the autoclose flag to "off."

```
<sdvs.1.3>  setflag
   flag variable:  autoclose
   on or off[on]:  off

   setflag autoclose -- off
```

We now check to ensure that *assign.temp.to.y.sd* is applicable and then apply it:

```
<sdvs.1.4>  applicable

 u(1) [sd pre: (true) comod: (all) mod: (y) post: (#y = .temp)]

<sdvs.1.4>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (y)
              post: (#y = .temp)]
```

The symbolic values of x and y should now meet the specification:

```
<sdvs.1.5>  ppeq
   expression:  .y

 eqclass = x\21

<sdvs.1.5>  ppeq
   expression:  .x

 eqclass = y\20
```

If the autoclose flag were "on," SDVS would have automatically closed the proof after the last application, because it knows that all of the goals have been met:

```
<sdvs.1.5>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.
```

Let us look at the proof state:

```
<sdvs.1.5>  ps

  << initial state >>
  proof in progress of example2.sd <5>
    apply (until #x = .y) <4>
    apply (until ...) <3>
    autoclose flag turned off <2>
    apply u(1) <1>
    --> you are here <--
```

Close the proof:

```
<sdvs.1.5>  close

 close -- 4 steps/applications
```

And see what **ps** has to say:

```
<sdvs.2>  ps

  << initial state >>
  proved example2.sd <1>
  --> you are here <--
```

At this point *example2.sd* is usable but not applicable. We could dump its proof via the **dump-proof** command or quit the proof via the **quit** command (which will end the proof session and associate the proof with the name "sdvsproof"). The command **dump-proof** does not end the proof session, i.e., after a **dump-proof**, the most recently proved state delta is still usable. Furthermore, **dump-proof** may be used in the middle of a proof. But **quit** may be used only at the end of a proof, and afterwards the state delta that was proved is no longer usable:

```
<sdvs.2>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  usable

 No usable state deltas.


 No usable quantified formulas.
```

A proof may also be pretty-printed:

```
<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  prove example2.sd
    proof:
      (until #x = .y,
       setflag autoclose off,
       apply u(1),
       close)
```

Let us turn the autoclose flag to "on" and prove *example2.sd* using the **until** command with the postcondition of *example2.sd* as its goal:

```
<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[on]:  on
```

```
 setflag autoclose -- on

<sdvs.2>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  example2.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x,y,temp),formula(assign.x.to.temp.sd))
          comod: (all)
            mod: (all)
           post: (#x = .y,#y = .x)]

 Complete the proof.

<sdvs.1.1>  until
   formula:  #x=.y and #y=.x

   apply -- [sd pre: (true)
             comod: (all)
               mod: (temp)
              post: (#temp = .x,formula(assign.y.to.x.sd))]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (x)
              post: (#x = .y,formula(assign.temp.to.y.sd))]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (y)
              post: (#y = .temp)]

 close -- 3 steps/applications
```

Now, **quit** will associate a different proof with "sdvsproof":

```
<sdvs.2>  quit
```

```
 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  prove example2.sd
    proof: until #x = .y & #y = .x
```

3.2 Proofs by Cases

During the course of a proof, a disjunction of two or more formulas may be true, and it may be that the proof can proceed only by considering each disjunct separately, i.e., it may be necessary to prove that each disjunct implies that the goal will be achieved. For this possibility, SDVS has the **cases** and **mcases** commands. Our next example will feature the use of the **cases** command.

Example 3 Consider the following conditional statement R:

```
if x <= y then
    z := y - x;
else
    z := x - y;
end if
```

At the end of the execution of this segment, the value of z should be the absolute value[8] of the difference of x and y, or equivalently, it should be true that

$$z \geq 0 \wedge (z = x - y \vee z = y - x)$$

The statement R may be translated in SDVS as the *conjunction* of the state delta *if.sd*

```
[sd pre: (.x le .y)
  comod: (all)
    mod: (z)
   post: (#z = .y - .x)]
```

and the state delta *else.sd*

```
[sd pre: (.y lt .x)
  comod: (all)
    mod: (z)
   post: (#z = .x - .y)]
```

Thus, the state delta *case.sd*

```
[sd pre: (covering(all,x,y,z),formula(if.sd),formula(else.sd))
  comod: (all)
    mod: (z)
   post: (#z ge 0,#z = .y - .x or #z = .x - .y)]
```

[8] SDVS has an absolute value function, abs, but in most cases, proofs that involve it require "reading" and invoking axioms, a subject that will be covered in Section 6.

asserts that the state delta translation of R implies its specification.

Let us initiate the proof of *case.sd*:

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[off]:  off

 setflag autoclose -- off

<sdvs.2>  prove
   state delta[]:  case.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x,y,z),formula(if.sd),formula(else.sd))
          comod: (all)
            mod: (z)
           post: (#z ge 0,
                  #z = .y - .x or #z = .x - .y)]

 Complete the proof.
```

The query **ppl** will display the symbolic values of the places x and y:

```
<sdvs.2.1>  ppl
   places[all]:  <CR>

 x  x\30
 y  y\29
```

The symbolic value of z is unimportant at this point, since $.z$ does not appear at the top level of the postcondition of *case.sd*.

The interpreted postcondition of *case.sd* is the goal of the proof:

```
<sdvs.2.1>  goals

 g(1) #z ge 0
 g(2) #z = y\29 - x\30 or #z = x\30 - y\29
```

And the state deltas *else.sd* and *if.sd* are usable:

```
<sdvs.2.1>  usable

 u(1) [sd pre: (.y lt .x)
       comod: (all)
         mod: (z)
        post: (#z = .x - .y)]

 u(2) [sd pre: (.x le .y)
       comod: (all)
         mod: (z)
        post: (#z = .y - .x)]


 No usable quantified formulas.
```

But they are not applicable:[9]

```
<sdvs.2.1>  applicable
```

The query **whynotapply** shows why:

```
<sdvs.2.1>  whynotapply
   state delta[ highest usable]:  if.sd

 Because the following is not known to be true -- .x le .y

<sdvs.2.1>  whynotapply
   state delta[ highest usable]:  else.sd

 Because the following is not known to be true -- .y lt .x
```

Neither is applicable because, at this state, neither precondition is true. But the disjunction of their preconditions is surely true:

```
<sdvs.2.1>  simp
   expression:  .x le .y or .y lt .x

 true
```

[9] If there are usable state deltas but none of which is known to be applicable by SDVS, the **applicable** query gives no information, because under certain circumstances, it may be possible for the user to prove that one of the usable state deltas is in fact applicable.

Since SDVS knows that the disjunction of these two formulas is true, we may use the **cases** command to split the proof of the goal to the case that $.x \leq .y$ and to the case that $.y < .x$. In the first case, *if.sd* will be applicable, and in the second case, *else.sd* will be applicable. But in both cases, the goal will remain the same, and so will the modification list. The comodification list will always be "all."

```
<sdvs.2.1> cases
   case predicate: .x le .y

   cases -- .x le .y

     open -- [sd pre: (.x le .y)
              comod: (all)
                mod: (z)
               post: (#z ge 0,
                      #z = y\29 - x\30 or #z = x\30 - y\29)]
```

The proof of the first case has been opened. The state has not been advanced (the "all" in the comodification list assures this), but it is now assumed that $.x \leq .y$:

```
<sdvs.2.1.1.1> simp
   expression: .x le .y

 true
```

As we already noted, the goal remains the same:

```
<sdvs.2.1.1.1> goals

 g(1) #z ge 0
 g(2) #z = y\29 - x\30 or #z = x\30 - y\29
```

Furthermore, since the state has not been advanced, the same state deltas are usable:

```
<sdvs.2.1.1.1> usable

 u(1) [sd pre: (.y lt .x)
       comod: (all)
         mod: (z)
        post: (#z = .x - .y)]

 u(2) [sd pre: (.x le .y)
       comod: (all)
         mod: (z)
```

```
        post: (#z = .y - .x)]

 No usable quantified formulas.
```

But in this case, since $.x \leq .y$, *if.sd* is also applicable:

```
<sdvs.2.1.1.1>  applicable

 u(2) [sd pre: (.x le .y)
       comod: (all)
         mod: (z)
        post: (#z = .y - .x)]
```

Let us see where we are in the proof, and then apply *if.sd*:

```
<sdvs.2.1.1.1>  ps

  << initial state >>
  autoclose flag turned off <3>
  proof in progress of case.sd <2>
    case analysis in progress on: .x le .y or ~(.x le .y) <1>
      1st case: in progress
       --> you are here <--

<sdvs.2.1.1.1>  apply
   sd/number[highest applicable/once]:  if.sd

       apply -- [sd pre: (.x le .y)
                 comod: (all)
                   mod: (z)
                  post: (#z = .y - .x)]
```

We inquire if the goal is true and close the proof of the first case:

```
<sdvs.2.1.1.2>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.2.1.1.2>  close

     close -- 1 steps/applications
```

```
    open -- [sd pre: (~(.x le .y))
             comod: (all)
               mod: (z)
              post: (#z ge 0,
                     #z = y\29 - x\30 or #z = x\30 - y\29)]

 Complete the proof.
```

SDVS has automatically opened the proof of the case $.y < .x$:

```
<sdvs.2.1.2.1>  simp
   expression:  .y lt .x

 true
```

At this point there are several usable state deltas, but only one has a true precondition and is applicable, *else.sd*:

```
<sdvs.2.1.2.1>  usable

 u(1) [sd pre: (.x le .y)
       comod: (all)
         mod: (z)
        post: (#z ge 0,
               #z = y\29 - x\30 or #z = x\30 - y\29)]

 u(2) [sd pre: (.y lt .x)
       comod: (all)
         mod: (z)
        post: (#z = .x - .y)]

 u(3) [sd pre: (.x le .y)
       comod: (all)
         mod: (z)
        post: (#z = .y - .x)]


 No usable quantified formulas.

<sdvs.2.1.2.1>  applicable

 u(2) [sd pre: (.y lt .x)
      comod: (all)
        mod: (z)
       post: (#z = .x - .y)]
```

Note that the first usable state delta is the state delta that was just proved, i.e., the first case state delta. We apply *else.sd* and close the proof of the second case:

```
<sdvs.2.1.2.1>  apply
   sd/number[highest applicable/once]:  else.sd

       apply -- [sd pre: (.y lt .x)
                 comod: (all)
                   mod: (z)
                  post: (#z = .x - .y)]
```

After the proof of the second case, SDVS automatically "joins" the two cases into one state delta and closes the proof of *case.sd*. (The "join" and close are automatic, even if the autoclose flag is off.) Thus *case.sd* is now usable:

```
<sdvs.3>  usable

 u(1) [sd pre: (covering(all,x,y,z),formula(if.sd),formula(else.sd))
        comod: (all)
          mod: (z)
         post: (#z ge 0,#z = .y - .x or #z = .x - .y)]


 No usable quantified formulas.
```

Let us quit and look at the proof:

```
<sdvs.3>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  (setflag autoclose off,
   prove case.sd
     proof:
       cases .x le .y
```

```
then proof:
  (apply if.sd,
   close)
else proof:
  (apply else.sd,
   close))
```

3.3 Proofs of Now and of Always

In this section, we will give some trivial examples of state deltas that assert that a formula is true now and that a formula is true always (in the timeline).

Example 4 A state delta with a comodification list of "all" and an empty modification list asserts that its precondition implies its postcondition at the current time. The reason for this is that a comodification list of "all" does not allow any variables to change value between now and the precondition time, and the empty modification list does not allow any variables to change value between the precondition and postcondition times. Thus, if the precondition implies the postcondition at the current state, the state delta is true.

Consider the state delta *now1.sd*:

```
[sd pre: (covering(all,x),.x gt a)
  comod: (all)
   post: (#x ge a + 1)]
```

It asserts[10] that $x > a$ implies $x \geq a + 1$, which is of course true. The proof is trivial.

```
<sdvs.4>  setflag
   flag variable:  autoclose
   on or off[on]:  off

 setflag autoclose -- off

<sdvs.5>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  now1.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x),.x gt a)
          comod: (all)
           post: (#x ge a + 1)]

 Complete the proof.
```

[10] Recall that in SDVS, *integer* is the default type of local and global variables.

Since the precondition has been asserted and the precondition implies the postcondition, the goal is true, and we may close the proof.

```
<sdvs.1.1>  simp
   expression:  .x gt a

 true

<sdvs.1.1>  simp
   expression:  .x ge a+1

 true

<sdvs.1.1>  close

 close -- 0 steps/applications
```

Example 5 A more interesting example of an implication posing as a state delta is *now2.sd*

```
  [sd pre: (covering(all,x),formula(event.x.gt.5.sd))
    comod: (all)
     post: (formula(event.x.ge.6.sd))]
```

where *event.x.gt.5.sd* is the state delta

```
  [sd pre: (true) comod: (all) mod: (x) post: (#x gt 5)]
```

and *event.x.ge.6.sd* is the state delta

```
  [sd pre: (true) comod: (all) mod: (x) post: (#x ge 6)]
```

Clearly, *event.x.gt.5.sd* implies *event.x.ge.6.sd*, which is the assertion of *now2.sd*.

Let us open the proof of this implication:

```
<sdvs.2>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.
```

```
<sdvs.1> prove
   state delta[]: now2.sd
   proof[]: <CR>

 open -- [sd pre: (covering(all,x),formula(event.x.gt.5.sd))
          comod: (all)
           post: (formula(event.x.ge.6.sd))]

 Complete the proof.
```

and look at the goals and the state deltas that are applicable:

```
<sdvs.1.1> goals

 g(1) [sd pre: (true) comod: (all) mod: (x) post: (#x ge 6)]
```

```
<sdvs.1.1> applicable

 u(1) [sd pre: (true) comod: (all) mod: (x) post: (#x gt 5)]
```

We really do not want to apply this last state delta right now, because

```
<sdvs.1.1> whynotapply
   state delta[ highest usable]: <CR>

 Applicable, but must lead to a contradiction, because modlist too large.
```

The problem is that the state delta that we are proving, *now2.sd*, has an empty modification list that does not allow us to advance the state under normal circumstances. To illustrate a point, we nevertheless proceed with the application:

```
<sdvs.1.1> apply
   sd/number[highest applicable/once]: <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (x)
              post: (#x gt 5)]

   Warning: the modlist of the last applied state delta mentions places (x)
   outside of the modlist of the state delta to be proven. The current
   proof can only be closed by contradiction.
```

SDVS will not allow us to close the proof unless we can prove that our application will eventually lead to a contradiction, i.e., an inconsistent state. If the postcondition of *event.x.gt.5.sd* did lead to a contradiction, for example, if the postcondition were $\#x = \#x + 1$, then we could close the proof. But since in this case the postcondition is not inconsistent, we must pop back one step.

```
<sdvs.1.2>  ps

  << initial state >>
  proof in progress of now2.sd <2>
    apply u(1) <1>
    --> you are here <--

<sdvs.1.2>  pop
   number of levels[1]:  <CR>

 One level popped.
```

The only way to proceed with the proof is to open the proof of *event.x.ge.6.sd*:

```
<sdvs.1.1>  goals

 g(1) [sd pre: (true) comod: (all) mod: (x) post: (#x ge 6)]

<sdvs.1.1>  prove
   state delta[]:  g
         number:  1
   proof[]:  <CR>

   open -- [sd pre: (true)
            comod: (all)
              mod: (x)
             post: (#x ge 6)]

 Complete the proof.


<sdvs.1.1.1>  applicable

 u(1) [sd pre: (true) comod: (all) mod: (x) post: (#x gt 5)]
```

At this point the modification list of the state delta to be proven, *event.x.g.6.sd*, is x, and this list is a sublist of the modification list of *event.x.gt.5.sd*. That is why we may apply it without having to reach a contradiction:

```
<sdvs.1.1.1>  whynotapply
   state delta[ highest usable]:  <CR>

 Quite applicable.

<sdvs.1.1.1>  apply
   sd/number[highest applicable/once]:  <CR>

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (x)
                post: (#x gt 5)]
```

To complete the proof we have to **close** twice, since we opened the proof of two state deltas:

```
<sdvs.1.1.2>  close

   close -- 1 steps/applications

 Complete the proof.

<sdvs.1.2>  usable

 u(1) [sd pre: (true) comod: (all) mod: (x) post: (#x ge 6)]

 u(2) [sd pre: (true) comod: (all) mod: (x) post: (#x gt 5)]


 No usable quantified formulas.

<sdvs.1.2>  close

 close -- 1 steps/applications

<sdvs.2>  usable

 u(1) [sd pre: (covering(all,x),formula(event.x.gt.5.sd))
       comod: (all)
        post: (formula(event.x.ge.6.sd))]


 No usable quantified formulas.
```

Example 6 A state delta whose comodification and modification lists are both empty asserts that, at every time in the future, the precondition implies the postcondition. The reason for this is that since the comodification list is empty, no constraint is made between the current time and any future time at which the precondition may be true. Thus, if at any future time the precondition is true, then — because the modification list is empty — the postcondition must be true at that very time. In particular, if the precondition is `true`, then the postcondition must be true now and at every future time.

To illustrate these remarks, we provide a simple example. The state delta *always.sd*

```
[sd pre: (true) post: (#x gt #y)]
```

asserts that the value of x is always greater than the value of y. Thus, *always.sd* in conjunction with the state delta *eventually1.sd*

```
[sd pre: (true) comod: (all) mod: (all) post: (#y = 100)]
```

implies the state delta *eventually2.sd*

```
[sd pre: (true) comod: (all) mod: (all) post: (#x gt 100)]
```

This implication is asserted by *always.ex.sd*

```
[sd pre: (covering(all,x,y),formula(eventually1.sd),formula(always.sd))
  comod: (all)
   post: (formula(eventually2.sd))]
```

which we proceed to prove.

```
<sdvs.3>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  always.ex.sd
   proof[]:  <CR>

 open -- [sd pre: (covering(all,x,y),formula(eventually1.sd),
                   formula(always.sd))
          comod: (all)
           post: (formula(eventually2.sd))]

 Complete the proof.
```

```
<sdvs.1.1>  usable

 u(1) [sd pre: (true) post: (#x gt #y)]

 u(2) [sd pre: (true)
        comod: (all)
          mod: (all)
         post: (#y = 100)]


 No usable quantified formulas.

<sdvs.1.1>  goals

 g(1) [sd pre: (true)
        comod: (all)
          mod: (all)
         post: (#x gt 100)]
```

There is only one efficient way to proceed:

```
<sdvs.1.1>  prove
   state delta[]:  g
         number:  1
   proof[]:  <CR>

   open -- [sd pre: (true)
             comod: (all)
               mod: (all)
              post: (#x gt 100)]

 Complete the proof.
```

Because the comodification list of *eventually2.sd* is `all`, *eventually1.sd* and *always.sd* are still usable (*always.sd* will always be usable, because its comodification list is empty, and it will always be applicable, because its precondition is `true` and its modification list is also empty.)

```
<sdvs.1.1.1>  usablesds

 u(1) [sd pre: (true) post: (#x gt #y)]

 u(2) [sd pre: (true)
        comod: (all)
          mod: (all)
```

```
        post: (#y = 100)]
```

We must now apply *eventually*1.*sd* and then *always.sd*, because the reverse order of application would not reach the goal. To illustrate this point, let us first apply *always.sd*:

```
<sdvs.1.1.1>  apply
   sd/number[highest applicable/once]:  always.sd

     apply -- [sd pre: (true)
                post: (#x gt #y)]

<sdvs.1.1.2>  simp
   expression:  .x gt .y

 true

<sdvs.1.1.2>  apply
   sd/number[highest applicable/once]:  eventually1.sd

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (all)
                post: (#y = 100)]

<sdvs.1.1.3>  simp
   expression:  .x gt .y

 x\51 gt 100
```

Because the modification list of *eventually*1.*sd* included x, the application of *eventually*1.*sd* erased the assertion $.x > .y$ from the data base of facts. So let us pop back, apply in the right order, and then close the two proofs.

```
<sdvs.1.1.3>  pop
   number of levels[1]:  2

 2 levels popped.

<sdvs.1.1.1>  apply
   sd/number[highest applicable/once]:  eventually1.sd

     apply -- [sd pre: (true)
               comod: (all)
                 mod: (all)
                post: (#y = 100)]
```

```
<sdvs.1.1.2>  apply
   sd/number[highest applicable/once]:  always.sd

     apply -- [sd pre: (true)
                 post: (#x gt #y)]

<sdvs.1.1.3>  close

   close -- 2 steps/applications

 Complete the proof.

<sdvs.1.2>  close

 close -- 1 steps/applications
```

3.4 Proofs by Induction

If a state delta is applicable at a certain point in a proof, and if its modification and comodification lists are disjoint, then the state delta may be applicable a number of times. For example, this is the case of the state delta

```
[sd pre: .i lt .y
  comod: x,y
    mod: s,i
   post: #s=.s+1 and #i=.i+1]
```

at time t_0 in the temporal structure M in Section 2.2.2. In certain situations it is possible to proceed with a proof by applying the state delta a fixed number of times, but in other cases the number of times that the state delta must be applied is not fixed but is data-dependent. For these instances, SDVS has a special proof command, **induct**. In this section we first present a simple and then a more complicated example illustrating this type of induction.

Example 7 Consider for example the state delta *x.increases.sd*:

```
[sd pre: (.x lt 100)
    mod: (x)
   post: (#x = .x + 1)]
```

If, at some point in a proof, *x.increases.sd* is true and the value of x is less than 100, then at this point *x.increases.sd* is not only true but applicable as well. In fact, from this point on, it may be applied repeatedly until the value of x reaches 100. Of course, the number of times that it must be applied for x to reach the value of 100 depends on the initial value of x itself. We will illustrate the **induct** command by giving a proof of the state delta *induction1.sd*:

```
[sd pre: (.x le 0,formula(x.increases.sd))
  comod: (all)
    mod: (x)
   post: (#x = 100)]
```

```
<sdvs.3> init
  proof name[]: <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1> prove
```

```
   state delta[]:  induction1.sd
   proof[]:  <CR>

 open -- [sd pre: (.x le 0,formula(x.increases.sd))
          comod: (all)
            mod: (x)
           post: (#x = 100)]

   inserting -- pcovering(all,x)

 Complete the proof.
```

The value of x is now symbolic and less than or equal to 0. In fact, let us give a name to this symbolic value by using a useful naming device in SDVS, the **let** command.

```
<sdvs.1.1>  let
   new variable:  a
          value:  .x

   let -- a = .x
```

It should be pointed out that the name of the "new variable" must be new and that the value assigned to it must be a term of type precondition (no # 's). For example,

```
<sdvs.1.2>  let
   new variable:  b
          value:  2*.x

   let -- b = 2 * .x

<sdvs.1.3>  simp
   expression:  b=2*a

 true
```

We can verify that a is the current value of x and list its range of possible values:

```
<sdvs.1.3>  ppeq
   expression:  .x

 eqclass = a
           x\54


<sdvs.1.3>  range
```

```
  expression: .x

Range -infinity ... 0
```

The state delta *x.increases.sd* is certainly applicable:

```
<sdvs.1.3> applicable

u(1) [sd pre: (.x lt 100)
        mod: (x)
       post: (#x = .x + 1)]
```

We are now ready to induct.

```
<sdvs.1.3> induct
    induction expression: i
                    from: a
                      to: 100
        invariant list[]: .x=i
   comodification list[]: <CR>
     modification list[]: x
           base proof[]: <CR>
           step proof[]: <CR>

   induction -- i from a to 100

     open -- [sd pre: (i = a)
               comod: (all)
                post: (.x = i)]
```

Several comments are in order:

(i) We are inducting from $i = a$ to $i = 100$.

(ii) The invariant list of the **induct** command must be true now. In general, the invariant list of the induction command is a precondition formula that must be true at every step of the induction, but not necessarily at every intermediate state.

(iii) If i does reach 100 and the invariant list is true, then our goal will also be true, namely, the value of x must then be 100.

(iv) The comodification and modification lists for the **induct** command must be disjoint.

(v) The modification list to the **induct** command must be included in the modification list of the state delta to be proven.

Upon the invocation of the induction command, SDVS automatically opens the proof of the base case state delta and then, upon the completion of this proof, the proof of the step case state delta. The base case proof is a proof that if i is equal to the initial value of $.x$, then the invariant is true now. The step case proof is the proof of the state delta that asserts that if i is somewhere between the specified limits of the range, i.e., $a \leq i < 100$, and the invariant is true for i, then there is a future time when the invariant will be true for $i + 1$, and in the meantime, only x may change its value. Since the autoclose flag is "off," we have to close the base case proof.

```
<sdvs.1.3.1.1>  close

     close -- 0 steps/applications

     open -- [sd pre: (i ge a,i lt 100,.x = i)
                 mod: (x)
                post: (#x = i + 1)]

 Complete the proof.
```

The proof of the state delta that was automatically opened is the step case proof. In order to complete this proof, we have to advance the state, and this can be done only by applying *x.increases.sd* (which, of course, is still applicable):

```
<sdvs.1.3.2.1>  applicable

 u(1) [sd pre: (.x lt 100)
         mod: (x)
        post: (#x = .x + 1)]

<sdvs.1.3.2.1>  apply
   sd/number[highest applicable/once]:  x.increases.sd

       apply -- [sd pre: (.x lt 100)
                   mod: (x)
                  post: (#x = .x + 1)]
```

Let us look at the proof state and then close the proof by induction:

```
<sdvs.1.3.2.2>  ps

  << initial state >>
  proof in progress of induction1.sd <5>
    let a = .x <4>
    let b = 2 * .x <3>
```

```
    induction in progress on i from a to 100 <2>
       base case: complete
       step case: in progress
        apply x.increases.sd <1>
        --> you are here <--

<sdvs.1.3.2.2>  close

     close -- 1 steps/applications

   join induction cases -- [sd pre: (a le 100)
                            comod: (all)
                              mod: (x)
                             post: (#x = 100)]

 Complete the proof.
```

Finally SDVS joins the two proofs. The goal should now have been reached.

```
<sdvs.1.4>  ppeq
   expression:  .x

 eqclass = 100


<sdvs.1.4>  close

 close -- 3 steps/applications
```

Our next example of induction in SDVS differs from the last one in that the invariant list for the induction and the comodification and modification lists is more complex.

Example 8 Suppose that at some point in the execution of a program, the value of the program variable y is greater than or equal to zero and the program segment to be executed is S:

```
sum:= x;
ctr:= 0;
while (ctr < y) loop
  sum:= sum + 1;
  ctr:= ctr + 1;
end loop;
```

At the end of the execution of S, the value of *sum* should be the sum of the initial values of x and y.

The loop portion of S cannot be translated into state deltas of the form that we have so far discussed. The semantics of a loop requires the concept of a circular state delta, which is defined as the greatest fixed point of an operator on predicates and is beyond the scope of this tutorial. But for another illustration of induction, we collapse the two assignment statements of the loop into one statement and translate the program segment S as the following series of nested state deltas:

assign.x.to.sum.sd:

```
[sd pre: (true)
  comod: (all)
    mod: (sum)
   post: (#sum = .x,formula(assign.0.to.ctr.sd))]
```

assign.0.to.ctr.sd:

```
[sd pre: (true)
  comod: (all)
    mod: (ctr)
   post: (#ctr = 0,formula(loop.sd))]
```

loop.sd:

```
[sd pre: (.ctr lt .y)
  comod: (x,y)
    mod: (sum,ctr)
   post: (#sum = .sum + 1,#ctr = .ctr + 1)]
```

The first two state deltas are assignment statements with a continuation. But *loop.sd* is the test for the loop (its precondition) and the collapsed loop body itself (its postcondition). Note that it is the only state delta without an "all" in its comodification list. When this state delta first becomes true, it will be applicable as long as x and y do not change their values and as long as the precondition is true. This is the state delta that will allow us to proceed with the induction.

Finally, the state delta *sum.sd*

```
[sd pre: (covering(all,x,y,ctr,sum),.y ge 0,formula(assign.x.to.sum.sd))
  comod: (all)
    mod: (sum,ctr)
   post: (#sum = .x + .y)]
```

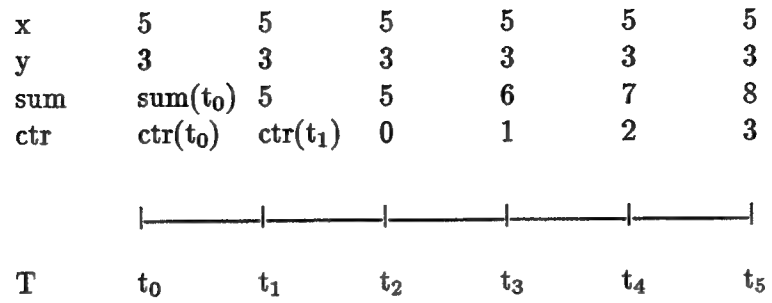

Figure 5: A Model N of the Precondition of *sum.sd*

asserts that if the initial value of y is greater than or equal to zero, and if *assign.x.to.sum* is true, then eventually the value of *sum* will be equal to the sum of the initial values of x and y. Furthermore, because of its modification list, in that interval of change x and y will remain constant.

For an example of a model N of the precondition of the state delta *sum.sd*, refer to Figure 5. In N the initial values of x and y are 5 and 3, respectively. The value of *sum* is symbolic at t_0, and the value of *ctr* is symbolic at t_0 and t_1. The state delta *assign.x.to.sum.sd* is applicable (true with a true precondition) at t_0; *assign.0.to.ctr.sd* is applicable at t_1; and *loop.sd* is applicable at t_2, t_3, and t_4. The precondition time of *sum.sd* is t_0, and its postcondition time is t_5.

Let us open the proof of *sum.sd*, check the symbolic values of x and y, and then check the goal of the proof:

```
<sdvs.3> init
   proof name[]: <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1> prove
   state delta[]: sum.sd
   proof[]: <CR>

 open -- [sd pre: (covering(all,x,y,ctr,sum),.y ge 0,
                   formula(assign.x.to.sum.sd))
           comod: (all)
             mod: (sum,ctr)
            post: (#sum = .x + .y)]

 Complete the proof.
```

```
<sdvs.1.1>  ppl
   places[all]:  <CR>

 x  x\62
 y  y\61

<sdvs.1.1>  goals

 g(1) #sum = x\62 + y\61
```

We now use **until** to advance through the first two assignment statements:

```
<sdvs.1.1>  until
   formula:  #ctr=0

   apply -- [sd pre: (true)
             comod: (all)
               mod: (sum)
              post: (#sum = .x,formula(assign.0.to.ctr.sd))]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (ctr)
              post: (#ctr = 0,formula(loop.sd))]

   until break point reached -- #ctr = 0
```

As we expected, the assignments have been executed and the goal remains the same:

```
<sdvs.1.3>  ppl
   places[all]:  <CR>

 sum  x\62
 everyplace  UNDEFINED
 ctr  0
 x  x\62
 y  y\61

<sdvs.1.3>  goals

 g(1) #sum = x\62 + y\61
```

The state delta *loop.sd* is now usable but not applicable because if the value of y happens to be 0, then its precondition is not true.

<sdvs.1.3> *usable*

```
u(1) [sd pre: (.ctr lt .y)
       comod: (x,y)
         mod: (sum,ctr)
        post: (#sum = .sum + 1,#ctr = .ctr + 1)]
```

```
No usable quantified formulas.
```

<sdvs.1.3> *applicable*

<sdvs.1.3> *whynotapply*
state delta[highest usable]: *loop.sd*

```
Because the following is not known to be true -- .ctr lt .y
```

We are thus forced to do cases on $.ctr \geq .y$. If this case is true, then $y = 0$ and our goal is trivially true:

<sdvs.1.3> *cases*
case predicate: *.ctr ge .y*

```
cases -- .ctr ge .y

  open -- [sd pre: (.ctr ge .y)
             comod: (all)
               mod: (sum,ctr)
              post: (#sum = x\62 + y\61)]
```

<sdvs.1.3.1.1> *close*

```
close -- 0 steps/applications

open -- [sd pre: (~(.ctr ge .y))
           comod: (all)
             mod: (sum,ctr)
            post: (#sum = x\62 + y\61)]
```

```
Complete the proof.
```

SDVS has now opened the case of $.ctr < .y$. Since this is the precondition of *loop.sd*, it is now applicable and we use it in our induction.

<sdvs.1.3.2.1> *applicable*

```
u(2) [sd pre: (.ctr lt .y)
      comod: (x,y)
        mod: (sum,ctr)
       post: (#sum = .sum + 1,#ctr = .ctr + 1)]
```

In this example the use of the **induct** command is less trivial. A little thought shows that $sum = x + ctr$ is true at the current state, and that it will continue to be true after each application of *loop.sd*. Furthermore, when the value of *ctr* is equal to the value of y, our goal will be true. It follows that we should induct from $.ctr = 0$ to $.ctr = .y$, with the invariant being $.sum = .x + .ctr$. The comodification list should consist of x and y, since they must remain constant during the induction, and the modification list should consist of *sum* and *ctr*, since they must be allowed to change. Note that these two lists are disjoint, as they must be. So, let us proceed with the induction.

```
<sdvs.1.3.2.1>  induct
    induction expression:  .ctr
                     from:  0
                       to:  .y
        invariant list[]:  .sum=.x+.ctr
   comodification list[]:  x,y
     modification list[]:  sum,ctr
           base proof[]:  <CR>
           step proof[]:  <CR>

      induction -- .ctr from 0 to .y

        open -- [sd pre: (true)
                 comod: (all)
                  post: (.sum = .x + .ctr,.ctr = 0)]
```

SDVS has opened the proof of the base case which is trivially true, and we close it.

```
<sdvs.1.3.2.1.1.1>  close

        close -- 0 steps/applications

        open -- [sd pre: (.ctr ge 0,.ctr lt .y,.sum = .x + .ctr)
                 comod: (x,y)
                   mod: (sum,ctr)
                  post: (#sum = #x + #ctr,#ctr = .ctr + 1)]

 Complete the proof.
```

For the step case proof, the invariant is assumed to be true now, and we have to reach the state at which it will continue to be true and the value of *ctr* will be incremented by 1. Let

us check the proof state at this juncture:

```
<sdvs.1.3.2.1.2.1>  ps

  << initial state >>
  proof in progress of sum.sd <5>
    apply (until #ctr = 0) <4>
    apply (until ...) <3>
    case analysis in progress on: .ctr ge .y or ~(.ctr ge .y) <2>
       1st case: complete
       2nd case: in progress
        induction in progress on .ctr from 0 to .y <1>
           base case: complete
           step case: in progress
            --> you are here <--
```

Since at this state $ctr < y$, and x and y have remained constant, *loop.sd* should be applicable:

```
<sdvs.1.3.2.1.2.1>  applicable

 u(1) [sd pre: (.ctr lt .y)
       comod: (x,y)
         mod: (sum,ctr)
        post: (#sum = .sum + 1,#ctr = .ctr + 1)]
```

Before applying this state delta, let us check the symbolic values of the places and the goal:

```
<sdvs.1.3.2.1.2.1>  ppl
   places[all]:  <CR>

 sum  sum\70
 everyplace  UNDEFINED
 ctr  ctr\69
 x  x\62
 y  y\61

<sdvs.1.3.2.1.2.1>  goals

 g(1) #sum = #x + #ctr
 g(2) #ctr = ctr\69 + 1
```

Now we apply *loop.sd* and check the symbolic values once more:

```
<sdvs.1.3.2.1.2.1>  apply
```

```
   sd/number[highest applicable/once]:  loop.sd

           apply -- [sd pre: (.ctr lt .y)
                     comod: (x,y)
                       mod: (sum,ctr)
                      post: (#sum = .sum + 1,#ctr = .ctr + 1)]

<sdvs.1.3.2.1.2.2>  ppl
   places[all]:  <CR>

 sum  (1 + x\62) + ctr\69
 everyplace  UNDEFINED
 ctr  1 + ctr\69
 x  x\62
 y  y\61
```

The values of *sum* and *ctr* have increased by one, and the invariant should remain true for these new values.

```
<sdvs.1.3.2.1.2.2>  simp
   expression:  .sum=.x+.ctr

 true

<sdvs.1.3.2.1.2.2>  applicable

<sdvs.1.3.2.1.2.2>  close

         close -- 1 steps/applications

       join induction cases -- [sd pre: (0 le .y)
                                comod: (all,x,y)
                                  mod: (sum,ctr)
                                 post: (#ctr = .y,#sum = #x + #y)]

 Complete the proof.
```

SDVS has now joined the two case of the induction proof. Let us check the proof state:

```
<sdvs.1.3.2.2>  ps

  << initial state >>
  proof in progress of sum.sd <5>
```

```
    apply (until #ctr = 0) <4>
    apply (until ...) <3>
    case analysis in progress on: .ctr ge .y or ~(.ctr ge .y) <2>
       1st case: complete
       2nd case: in progress
        proved via induction, then applied
          [sd pre: (0 le .y)
            comod: (all,x,y)
              mod: (sum,ctr)
             post: (#ctr = .y,#sum = #x + #y)] <1>
        --> you are here <--
```

The goal of the second case of the cases true has now been reached.

```
<sdvs.1.3.2.2>  simp
   expression:  .sum=.x+.y

 true
```

We thus close the second case proof and then the proof of *sum.sd* itself.

```
<sdvs.1.3.2.2>  close

     close -- 1 steps/applications

   join -- [sd pre: (true)
             comod: (all)
               mod: (sum,ctr)
              post: (#sum = x\62 + y\61)]

 close -- 3 steps/applications

<sdvs.2>  ps

  << initial state >>
  proved sum.sd <1>
  --> you are here <--

<sdvs.2>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.
```

4 Declaration of Types

In our previous examples we did not declare the type of the variables. SDVS assumed that they were integers. However, it is possible to declare explicitly the types of the local variables by the **declare** statement. The SDVS types accepted by the **declare** statement may be listed with the **help** query:

```
<sdvs.1> help
   with[all]: types


 <<<SDVS Help>>>   Types   <<<SDVS Help>>>

 type(boolean)  Boolean

 type(character)  Ada characters

 type(bitstring,n)  bitstring of length n

 type(polymorphic)  polymorphic (any type)

 type(fn,exp)  a function defined by the expression exp

 type(float)  floating point

 type(integer)  integer

 type(integer,lb,ub)  bounded integer, that is, lb<=i<=ub

 type(array,lb,ub,type)  array with lower bound lb, upper bound ub, and
                         specified element type

 type(record,field1(type1),...,fieldj(typej))  record with field names of
                                               specified types

 type(time)  VHDL time

 type(waveform)  VHDL waveform

 type(integerwaveform)  VHDL integer waveform

 type(bitwaveform)  VHDL bit waveform

 type(bitstringwaveform,n)  VHDL bitstring (length n) waveform
```

The last five types are peculiar to VHDL. The "record," "character," and "float" types are for Ada, and the bitstring type is for ISPS and VHDL. If a is an array, then $origin(a)$ is its initial index, $range(a)$ is its length, and for i and j such that $origin(a) \leq i \leq j \leq (origin(a) + range(a))$, $a[i:j]$ is the slice (subarray) of a from the i'th to the j'th index. If b is a bitstring, then $lh(b)$ is the length of b, the zeroth bit is the low-order bit, and the $lh(b) - 1$ bit is the high-order bit. The integer value of b is denoted by $|b|$. If i and j are such that $0 \leq j$ and $0 \leq i \leq 2^j - 1$, then $i(j)$ is the bitstring of length j and integer value i. Thus, if $b = 10(4)$, then $b =< 1010 >$. If $0 \leq i \leq j \leq lh(b) - 1$, then $b < j : i >$ is the substring of b of length $j - i + 1$ whose bits are the bits of b from the i'th to the j'th bit. Some bitstring operations are addition (++), subtraction (-), multiplication (**), bitstring "or" (usor), bitstring "and" (&&), and concatenation (@). Some bitstring inequalities are: "uslt," "usle," "usgt," and "usge" (the "us" prefix means unsigned). Here is an example of a few declarations:

```
<sdvs.1>  pp
   object:  sd
   state delta name:  types.sd

 [sd pre: (declare(a,type(array,-1,5,type(integer))),
           declare(abit,type(array,1,10,type(bitstring,4))),
           declare(p,type(boolean)),declare(q,type(boolean)),
           declare(r,type(boolean)),declare(cbit,type(bitstring,6)),
           covering(all,a,abit,p,q,r),.p,~.q,.r,.a[1] = 2,.a[2] = 100,
           .abit[1] = 9(4),.abit[2] = 8(4),.abit[3] = 7(4),
           .cbit = 12(6))
   comod: (all)
     mod: (p,q)
    post: (#q)]
```

Note that the initial values of $abit[1]$, $abit[2]$, and $abit[3]$ are $< 1001 >$, $< 1000 >$, and $< 0111 >$, respectively. Also, initially, p and r are true, and q is false. The above state delta is, of course, not provable, but if we open its proof we can then use the simplifier to illustrate some notation:

```
<sdvs.1>  prove
   state delta[]:  types.sd
   proof[]:  <CR>

 open -- [sd pre: (declare(a,type(array,-1,5,type(integer))),
                   declare(abit,type(array,1,10,type(bitstring,4))),
                   declare(p,type(boolean)),declare(q,type(boolean)),
                   declare(r,type(boolean)),
                   declare(cbit,type(bitstring,6)),
                   covering(all,a,abit,p,q,r),.p,~.q,.r,.a[1] = 2,
                   .a[2] = 100,.abit[1] = 9(4),.abit[2] = 8(4),
```

```
                .abit[3] = 7(4),.cbit = 12(6))
          comod: (all)
            mod: (p,q)
           post: (#q)]

  inserting -- pcovering(all,cbit)

Complete the proof.

<sdvs.1.1> simp
  expression: .q

false

<sdvs.1.1> simp
  expression: .q implies (.p and .r)

true

<sdvs.1.1> simp
  expression: .q or (.q implies (.p and .r))

true

<sdvs.1.1> simp
  expression: .a[1]+.a[2]

102

<sdvs.1.1> simp
  expression: origin(a)

-1

<sdvs.1.1> simp
  expression: range(a)

7

<sdvs.1.1> simp
  expression: range(a[2:3])

2

<sdvs.1.1> simp
```

```
   expression:  .a[2:3][2]

 100

<sdvs.1.1>  simp
   expression:  |.abit[2]|

 8

<sdvs.1.1>  simp
   expression:  .abit[2]

 8(4)

<sdvs.1.1>  simp
   expression:  lh(.cbit)

 6

<sdvs.1.1>  simp
   expression:  .abit[1]++.abit[2]

 17(5)

<sdvs.1.1>  simp
   expression:  .cbit<2:0>

 4(3)

<sdvs.1.1>  simp
   expression:  .cbit<3:3>

 1(1)

<sdvs.1.1>  simp
   expression:  .abit[1] usor .abit[2]

 9(4)

<sdvs.1.1>  simp
   expression:  .abit[1] @ .abit[2]

 152(8)
```

5 Quantification in SDVS

Quantification and proof rules involving quantifiers have been implemented in SDVS, but not in a very general way. The universal quantifier $\forall$ is "forall" and the existential quantifier $\exists$ is "exists." Both of these quantifiers may be used, untyped, over values of program variables (places), but only the existential quantifier may be used over the program variables themselves. In this section we illustrate two of the most important quantification proof rules, **instantiate**, and **provebyinstantiation**, by using a simple example.

Example 9 Consider the state delta *quant.sd*

```
[sd pre: (declare(a,type(array,1,10,type(integer))),
          forall i (1 le i & i le 10 --> .a[i] = 1),
          exists j ((1 le j & j le 10) & formula(increase.aj.sd)))
  comod: (all)
    mod: (all)
   post: (exists k (#a[k] = 3))]
```

where *increase.aj.sd* is the state delta

```
[sd pre: (true)
    mod: (a[j])
   post: (#a[j] = .a[j] + 1)]
```

The precondition of *quant.sds* declares a to be an array variable of ten integers such that, initially, all the indexed values of a are equal to 1. It also asserts that for some index j in the range of indices of a, *increase.aj.sd* is initially true. Because the comodification list of *increase.aj.sd* is the empty list, *increase.aj.sd* asserts that from now on, for this index j and for whatever value $a[j]$ has, there will be a future time when this value will increase by 1. Thus, *quant.sd* asserts that if its precondition is now true, then there will be a time in the future at which the value of some $a[k]$ will be equal to 3. Of course, one of these k's will be j.

The proof of *quant.sd* requires both of the quantification proof rules mentioned above; moreover, it requires the use of **instantiate** in two different contexts.

```
<sdvs.1>  prove
   state delta□:  quant.sd
   proof□:  <CR>

 open -- [sd pre: (declare(a,type(array,1,10,type(integer))),
                   forall i (1 le i & i le 10 --> .a[i] = 1),
                   exists j ((1 le j & j le 10) &
```

```
                                formula(increase.aj.sd)))
            comod: (all)
              mod: (all)
             post: (exists k (#a[k] = 3))]

 Complete the proof.

<sdvs.1.1>  setflag
   flag variable:  autoclose
   on or off[on]:  off

   setflag autoclose -- off

<sdvs.1.2>  usable

 No usable state deltas.


 q(1) exists j ((1 le j & j le 10) &
                ([sd pre: (true)
                     mod: (a[j])
                    post: (#a[j] = .a[j] + 1)]))

 q(2) forall i (1 le i & i le 10 --> .a[i] = 1)

<sdvs.1.2>  goals

 g(1) exists k (#a[k] = 3)
```

Notice that the query **usable** lists two quantified statements, one of which involves the state delta *increase.aj.sd*, and that the goal of the proof is an existentially quantified statement. The quantified assertion $q(1)$ can not be applied because it is not a state delta: we must first instantiate the j in $q(1)$ by some variable that has not appeared outside of $q(1)$; in fact, we will use j itself for the substitution. (This technique is similar to a technique in predicate logic that consists of substituting a new constant in a formula for an existentially quantified variable of the formula). The proof command **instantiate** allows us to perform this substitution and delete the existential quantifier from $q(1)$. Note that, in this case, we are "instantiating" a usable formula to remove the existential quantifier. Later in the proof, we will use **instantiate** to prove the existentially quantified goal.

```
<sdvs.1.2>  instantiate
      existential formula:  q
                   number:  1
   existential variable[]:  j
         instantiated by:  j
```

```
   existential variable[]: <CR>

  instantiate in q(1) -- j for j.
```

```
<sdvs.1.3> usable

u(1) [sd pre: (true)
        mod: (a[j])
       post: (#a[j] = .a[j] + 1)]


q(1) exists j ((1 le j & j le 10) &
                ([sd pre: (true)
                     mod: (a[j])
                    post: (#a[j] = .a[j] + 1)]))

q(2) forall i (1 le i & i le 10 --> .a[i] = 1)
```

In this case the parameters to the **instantiate** command are the usable existential formula $q(1)$, the existential variable, and the variable by which it will be replaced in the matrix of $q(1)$, followed by a carriage return to the last query.

If the formula ϕ to be instantiated has a series of existential variables in its prefix, i.e., if ϕ is of the form $(\exists x_{i_1})(\exists x_{i_2})\ldots(\exists x_{i_n})\psi$, then only one invocation of the **instantiate** command is needed to instantiate all of the x_i's. The parameters to the **instantiate** command would be the formula to be instantiated, followed by the first existential variable and the variable by which it will be replaced, followed by the second existential variable and the variable by which it will be replaced, etc. The input to the command terminates with a carriage return to the query "existential variable[]".

Note that the instantiated formula *increase.aj.sd* is now usable and, of course, applicable. We will apply it twice. But first we must establish that, at this point, $a[j] = 1$. This follows logically from $q(2)$, but SDVS is not automatically aware of it. It may be established by the **provebyinstantiation** command. If in the course of a proof a formula of the form $(\forall x)\phi(x)$ is usable, **provebyinstantiation** may be used to assert $\phi(c/x)$, for any term c. More generally, if $(\forall x_{i_1})(\forall x_{i_2})\ldots(\forall x_{i_n})\phi$ is usable, then one invocation of **provebyinstantiation** suffices to assert $\phi(c_{i_1}/x_{i_1}, c_{i_2}/x_{i_2}\ldots c_{i_n}/x_{i_n})$.

```
<sdvs.1.3> provebyinstantiation
          prove formula[]: .a[j]=1
   using universal formula: q
                    number: 2
      universal variable[]: i
           instantiated by: j
      universal variable[]: <CR>
```

```
   provebyinstantiation -- a\90 = 1

<sdvs.1.4>  simp
   expression:  .a[j]

 1
```

Before we proceed, we should note that we could not have switched the last two proof commands. If we had first asserted that $a[j] = 1$ and then tried to instantiate, an error would have occurred in the instantiation, since at that point j would have already been used. If we apply *increase.aj.sd* twice, we will reach the state at which our goal is true.

```
<sdvs.1.4>  apply
   sd/number[highest applicable/once]:  2

   apply -- [sd pre: (true)
                mod: (a[j])
               post: (#a[j] = .a[j] + 1)]

   apply -- [sd pre: (true)
                mod: (a[j])
               post: (#a[j] = .a[j] + 1)]

<sdvs.1.6>  ps

  << initial state >>
  proof in progress of quant.sd <6>
    autoclose flag turned off <5>
    instantiate j for j in q(1) <4>
    provebyinstantiation .a[j] = 1 <3>
    apply <2>
    apply <1>
    --> you are here <--

<sdvs.1.6>  simp
   expression:  .a[j]

 3
```

So our goal is true, but again SDVS does not automatically know that $a[j] = 3$ implies that $(\exists k)(a[k] = 3)$.

```
<sdvs.1.6> whynotgoal
   simplify?[no]: <CR>

 g(1) exists k (#a[k] = 3)
```

We must prove this through another invocation of the **instantiate** command, where this time the existentially quantified formula is not a usable formula but a goal. If $(\exists x)\phi(x)$ is a goal and we invoke **instantiate** with this goal as the existentially quantified formula, x as the existential variable, and c as the term to be substituted in ϕ for x, then the goal becomes $\phi(c/x)$.

```
<sdvs.1.6> instantiate
       existential formula: g
                    number: 1
    existential variable[]: k
          instantiated by: j
    existential variable[]: <CR>

   instantiate in goal 1 -- j for k.

<sdvs.1.7> goals

 g(1) #a[j] = 3
```

SDVS knows that this goal is true; so we close the proof and quit:

```
<sdvs.1.7> close

 close -- 6 steps/applications

<sdvs.2> quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1> pp
   object: proof
   proof name: sdvsproof

 proof sdvsproof:
```

```
prove quant.sd
  proof:
    (setflag autoclose off,
     instantiate (j=j) in q(1),
     provebyinstantiation .a[j] = 1
       using: q(2)
       substitutions: (i=j),
     apply 2,
     instantiate (k=j) in g(1),
     close)
```

6 Static Proofs

In actual practice the most difficult and time-consuming parts of a proof of program correctness in SDVS are not the dynamic but rather the static parts of the proof, that is, those parts of the proof that require the proof of a nontemporal formula that the simplifier does not automatically know to be true.

The reason for this is that complete decision procedures have been implemented in the simplifier for only certain theories over the SDVS domains.

For example, complete decision procedures have been implemented for propositional logic and for quantifier-free formulas for integer and bitstring arithmetic with addition, but not for integer or bitstring arithmetic that involves multiplication or for integer arithmetic that involves the absolute-value function and the integer logarithmic function to the base 2. A few facts are known by the simplifier about these operations, but certainly not all. For example, the simplifier knows[11] that the following statements are true:

```
<sdvs.1> simp
   expression: x+y=y+x

 true

<sdvs.1> simp
   expression: 2*x=x*2

 true

<sdvs.1> simp
   expression: 3*(x+y-2*z)=3*y-6*z+3*x

 true

<sdvs.1> simp
   expression: a lt 0 implies abs(a)=-a

 true

<sdvs.1> simp
   expression: a ge 0 implies abs(a)=a

 true

<sdvs.1> simp
```

[11] By "knows", we mean that the statement in question is automatically derivable from the current state by the simplifier.

```
   expression:  i le 127 and j le 127 and a=i(7) and b=j(7) implies |a++b|=|b++a|

 true
```

But it does not automatically know that these are also true:

```
<sdvs.1>  simp
   expression:  x*y=y*x

 x * y = y * x

<sdvs.1>  simp
   expression:  x*(x+2)=x*x+2*x

 x * (2 + x) = 2 * x + x * x

<sdvs.1>  simp
   expression:  abs(a-b)=abs(b-a)

 abs(a - b) = abs(b - a)

<sdvs.1>  simp
   expression:  i le 127 and j le 127 and a=i(7) and b=j(7) implies |a**b|=|b**a|

 i le 127 & (j le 127 & (a = i(7) & b = j(7)))
    --> |a ** b| = |b ** a|
```

The latter propositions must be proved in the larger context of a proof of a state delta, by invoking the appropriate SDVS axioms. Alternatively, the user may create and prove a lemma and then apply it in the course of a larger proof. This alternative is especially useful if the lemma is required at more than one point in a proof. It is even possible for a lemma to be created and used in the proof of a state delta and the proof of the lemma to be deferred for a later time. At the end of the proof of a state delta (after a **quit**), SDVS will inform the user of those lemmas that were used but not proved prior to the proof.

SDVS axioms are invoked by means of the **rewritebyaxiom** and **provebyaxiom** commands. Before an axiom name is entered as a parameter to these proof commands, the SDVS axiom file that contains it must first be read by the **read** command. The SDVS axiom files may be listed by the **help** command:

```
<sdvs.1>  help
   with[all]:  axioms


 <<<SDVS Help>>>   Axioms   <<<SDVS Help>>>
```

```
axioms/abs.axioms  integer absolute value

axioms/arraycoverings.axioms  arrays and coverings

axioms/arrays.axioms  0-origin arrays (obsolete)

axioms/bitstring.axioms  bitstrings

axioms/div.axioms  integer division

axioms/exp.axioms  integer exponentiation

axioms/idiv.axioms  unsigned integer division

axioms/lastone.axioms  the LAST.ONE bitstring function

axioms/log2.axioms  integer log base 2

axioms/minmax.axioms  integer min and max

axioms/mod.axioms  integer modulus

axioms/mult.axioms  integer multiplication

axioms/origin-arrays.axioms  arbitrary-origin arrays

axioms/quant.axioms  quantification

axioms/rem.axioms  integer remainder

axioms/sqrt.axioms  integer square root
```

They may be **read**:

```
<sdvs.1>  read
   path name[tutorial/example2]:  axioms/mult.axioms

 Definitions read from file "axioms/mult.axioms"
   -- (mult0,mult1,multcommute,multassoc,multdistributeplus,
       multdistributeminus,multminus,multsquarege0,multge0,multle0,multge,
       multgt0,multlt0,multgt)

<sdvs.2>  read
   path name[axioms/mult.axioms]:  axioms/abs.axioms
```

```
Definitions read from file "axioms/abs.axioms"
  -- (abs\neg,abs\pos)

<sdvs.3>  read
   path name[axioms/abs.axioms]:  axioms/exp.axioms

Definitions read from file "axioms/exp.axioms"
  -- (e1,e2,e3,e4,e5,e6,e7,expmult,expdiv,e8,e9,e10,e11,expabsval,
      multeqsquare)
```

Each axiom that has been read may be pretty-printed:

```
<sdvs.4>  pp
   object:  axiom
   axiom name:  multdistributeplus

axiom multdistributeplus (x,y,z):
          x * (y + z) = x * y + x * z
```

A list of the current axioms (i.e., the axioms that have been read in the current session, that contain a specific function or predicate symbol) may also be displayed with the **pp** command. The "symbol" that is entered as a parameter to the **pp** command must be the simplifier name for that symbol. The simplifier symbols may be listed with the **help** command:

```
<sdvs.4>  help
   with[all]:  symbols


<<<SDVS Help>>>   Symbols used in Axioms and Lemmas   <<<SDVS Help>>>

constants  everyplace, emptyplace, emptyarray, true, false

functions  mkarray, val, inertial_update, transport_update, transaction,
           waveform, abs, mod, rem, div (/), min, max, expt (^), mult (*),
           minus (-), plus (+), parity, lastone, ones, zeros, useqv,
           usnor, usnand, usxor, usor, usand (&&), usnot (~~), usremainder
           (usmod), usquotient (//), ustimes (**), usdifference (--),
           usplus (++), usgeq (usge), usgtr (usgt), usleq (usle), uslss
           (uslt), usneq (~==), useql (==), usconc (@), ussub, bcons, bs,
           usval, lh, aconc, element, origin, range, slice, union, diff,
           vhdltime, timeglobal, timedelta, timeplus, tcval

predicates  timege, timegt, timele, timelt, vhdltimep, sd-value, lt, le,
```

```
gt, ge, alldisjoint, pcovering, covering, disjointarray, lhp,
usvalp, elt, ele, egt, ege, esucc, epred, cond, and (&), or,
xor, implies (-->), not (~), eq (=), neq (~=), distinct,
preemption, waveformp
```

Here is a list of the current axioms with the "minus" function:

```
<sdvs.4>  pp
   object:  axioms
   axiom names[]:  <CR>
   with symbols[]:  minus

 axiom expabsval (a,b,c):  ((b ge a & a ge -b) & b ge 0) & c ge 1
                      --> b ^ c ge a ^ c

 axiom expdiv (a,k):  k ge 1 --> a ^ (k - 1) = a ^ k / a

 axiom expmult (a,k):  k ge 1 --> a ^ k = a * a ^ (k - 1)

 axiom abs\neg (x):  x lt 0 --> abs(x) = -x

 axiom multminus (x,y):  (-x) * y = -(x * y)

 axiom multdistributeminus (x,y,z):  x * (y - z) = x * y - x * z
```

The **pp** command may also be used to display the current axioms that contain a list of specified symbols. Here is how to list the current axioms in which both "^" and "-" appear:

```
<sdvs.4>  pp
   object:  axioms
   axiom names[]:  <CR>
   with symbols[]:  expt,minus

 axiom expabsval (a,b,c):  ((b ge a & a ge -b) & b ge 0) & c ge 1
                      --> b ^ c ge a ^ c

 axiom expdiv (a,k):  k ge 1 --> a ^ (k - 1) = a ^ k / a

 axiom expmult (a,k):  k ge 1 --> a ^ k = a * a ^ (k - 1)
```

Furthermore, there is a command to delete all or some of the current axioms:

```
<sdvs.4>  deleteaxioms
   axiom names[all]:  <CR>
```

```
deleteaxioms -- (e10,e9,e7,e6,e1,multgt,multlt0,multgt0,expabsval,e11,e8,
                 e2,multge,multle0,multge0,multsquarege0,multeqsquare,
                 expdiv,expmult,e5,e4,e3,abs\pos,abs\neg,multminus,
                 multdistributeminus,multdistributeplus,multassoc,
                 multcommute,mult1,mult0)
```

6.1 Invoking SDVS Axioms

Example 10 In this example we prove a simple mathematical formula, due to Gauss, that is translated as a state delta loop. The formula states that the sum of the first n positive integers is equal to $n(n+1)/2$, i.e., $\sum_{i=1}^{n} i = n(n+1)/2$. To simplify matters we prove the equivalent assertion $2 * \sum_{i=1}^{n} i = n(n+1)$.

The sum can be calculated by a "while" loop. One interpretation of the formula is given by the state delta *gauss.sd*

```
[sd pre: (covering(all,i,sigma),.i = 1,n ge 1,.sigma = 1,
          formula(gaussloop.sd))
  comod: (all)
    mod: (i,sigma)
   post: (2 * #sigma = n * (n + 1),#i = n)]
```

where *gaussloop.sd* is the state delta

```
[sd pre: (.i lt n)
    mod: (sigma,i)
   post: (#i = .i + 1,#sigma = .sigma + #i)]
```

Not surprisingly, the SDVS proof of *gauss.sd* mirrors the standard proof by induction of the formula.

```
<sdvs.4> init
   proof name[]: <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1> setflag
   flag variable: autoclose
   on or off[on]: off
```

```
setflag autoclose -- off

<sdvs.2>  prove
   state delta□:  gauss.sd
   proof□:  <CR>

 open -- [sd pre: (covering(all,i,sigma),.i = 1,n ge 1,.sigma = 1,
                   formula(gaussloop.sd))
          comod: (all)
            mod: (i,sigma)
           post: (2 * #sigma = n * (n + 1),#i = n)]

 Complete the proof.


<sdvs.2.1>  induct
    induction expression:  .i
                    from:  1
                      to:  n
        invariant list□:  2*.sigma=.i*(.i+1)
   comodification list□:  <CR>
     modification list□:  i,sigma
           base proof□:  <CR>
           step proof□:  <CR>

   induction -- .i from 1 to n

     open -- [sd pre: (true)
              comod: (all)
               post: (2 * .sigma = .i * (.i + 1),.i = 1)]


<sdvs.2.1.1.1>  close

     close -- 0 steps/applications

     open -- [sd pre: (.i ge 1,.i lt n,
                       2 * .sigma = .i * (.i + 1))
                mod: (i,sigma)
               post: (2 * #sigma = #i * (#i + 1),#i = .i + 1)]

 Complete the proof.
```

SDVS has opened the step case of the induction proof. The precondition of the step case state delta is the assumption that the formula is true for $.i$, and the postcondition is the assertion that the formula is true for $.i + 1$. If a decision procedure for multiplication of

integers had been implemented in the simplifier, then an application of the applicable state delta *gaussloop.sd* would close the proof. But as we have pointed out, this is not the case: we will have to "read" and use the SDVS axioms for integer multiplication.

For a clear analysis of our strategy, we simplify matters by first naming the current values of i and *sigma*, since, after the application of *gaussloop.sd*, these values will change. (Actually, these values have a symbolic representation that will not be erased by the application of *gaussloop.sd*, but it is more elegant to give them names.)

```
<sdvs.2.1.2.1>  let
   new variable:  oldi
           value:  .i

       let -- oldi = .i

<sdvs.2.1.2.2>  let
   new variable:  oldsigma
           value:  .sigma

       let -- oldsigma = .sigma

<sdvs.2.1.2.3>  ppeq
   expression:  .i

 eqclass = oldi
           i\103

<sdvs.2.1.2.3>  ppeq
   expression:  .sigma

 eqclass = oldsigma
           sigma\104

<sdvs.2.1.2.3>  apply
   sd/number[highest applicable/once]:  gaussloop.sd

       apply -- [sd pre: (.i lt n)
                    mod: (sigma,i)
                   post: (#i = .i + 1,#sigma = .sigma + #i)]
```

At this point, the simplifier should know that the following equalities are true:

EQ 1: $2 * oldsigma = oldi * (oldi + 1)$

EQ 2: $.i = oldi + 1$

EQ 3: $.sigma = oldsigma + (oldi + 1)$

Furthermore, the current goal

$$2 * .sigma = .i * (.i + 1)$$

is known by SDVS as the equivalent equation

$$E: \qquad 2 * (oldsigma + (oldi + 1)) = (oldi + 1) * (oldi + 2)$$

Let us verify these assertions:

```
<sdvs.2.1.2.4>  simp
   expression:  2*oldsigma=oldi*(oldi+1)

 true

<sdvs.2.1.2.4>  simp
   expression:  .i=oldi+1

 true

<sdvs.2.1.2.4>  simp
   expression:  .sigma=oldsigma+(oldi+1)

 true

<sdvs.2.1.2.4>  whynotgoal
   simplify?[no]:  yes

 g(1) 2 * ((1 + oldi) + oldsigma) = (1 + oldi) * (2 + oldi)
```

If we substitute the right-hand side of EQ 1 for the term $2 * oldsigma$ in equation E and expand the terms $oldi * (oldi + 1)$ and $(oldi + 1) * (oldi + 2)$, we see that this new equation $E1$ is in fact true. Yet the simplifier does not know the goal to be true:

```
<sdvs.2.1.2.4>  whynotgoal
   simplify?[no]:  <CR>

 g(1) 2 * #sigma = #i * (#i + 1)
```

The reason that SDVS does not know that the goal is true is that, although the substitution is done automatically, the expansions are not, and the simplifier does not know automatically that the expansions are true:

```
<sdvs.2.1.2.4>  simp
   expression:  oldi*(oldi+1)=oldi*oldi+oldi

 2 * oldsigma = oldi + oldi * oldi

<sdvs.2.1.2.4>  simp
   expression:  (oldi+1)*(oldi+2)=oldi*oldi+3*oldi+2

 (1 + oldi) * (2 + oldi) = (2 + oldi * oldi) + 3 * oldi
```

This is why we must use the SDVS integer-multiplication axioms. But first, let us dump our partial proof so that we may use it in the next example:

```
<sdvs.2.1.2.4>  dump-proof
   name:  gauss.partial1.proof

 Current proof dumped to gauss.partial1.proof.
```

We now "read" the integer multiplication axioms and pretty-print the two that we will need.

```
<sdvs.2.1.2.4>  read
   path name[axioms/exp.axioms]:  axioms/mult.axioms

       Definitions read from file "axioms/mult.axioms"
         -- (mult0,mult1,multcommute,multassoc,multdistributeplus,
             multdistributeminus,multminus,multsquarege0,multge0,multle0,
             multge,multgt0,multlt0,multgt)


<sdvs.2.1.2.5>  pp
   object:  axiom
   axiom name:  multdistributeplus

 axiom multdistributeplus (x,y,z):
           x * (y + z) = x * y + x * z
<sdvs.2.1.2.5>  pp
   object:  axiom
   axiom name:  multcommute

 axiom multcommute (x,y):
           x * y = y * x
```

If at a certain point in a proof we wish to assert that two terms t_1 and t_2 are equal and a current axiom has the form $t_1^* = t_2^*$ or $t_2^* = t_1^*$, where t_1 and t_2 are of the form t_1^* and t_2^*, respectively, then an invocation of the **rewritebyaxiom** command with the current axiom

as the "axiom name" parameter and the term t_1 as the "term to rewrite" parameter will assert the equality of t_1 and t_2.[12]

We first expand the term $oldi * (oldi + 1)$ by means of the **rewritebyaxiom** command.

```
<sdvs.2.1.2.5>  rewritebyaxiom
   term to rewrite:  oldi*(oldi+1)
      axiom name[]:  multdistributeplus

       rewritebyaxiom multdistributeplus -- oldi * (oldi + 1)
                                          = oldi * oldi +
                                              oldi * 1
```

```
<sdvs.2.1.2.6>  simp
   expression:  oldi*(oldi+1)=oldi*oldi+oldi

 true
```

The expansion of $(oldi + 1) * (oldi + 2)$ is more complicated:

```
<sdvs.2.1.2.6>  rewritebyaxiom
   term to rewrite:  (oldi+1)*(oldi+2)
      axiom name[]:  multdistributeplus

       rewritebyaxiom multdistributeplus -- (oldi + 1) * (oldi + 2)
                                          = (oldi + 1) * oldi +
                                              (oldi + 1) * 2
```

```
<sdvs.2.1.2.7>  rewritebyaxiom
   term to rewrite:  (oldi+1)*oldi
      axiom name[]:  multcommute

       rewritebyaxiom multcommute -- (oldi + 1) * oldi
                                   = oldi * (oldi + 1)
```

```
<sdvs.2.1.2.8>  rewritebyaxiom
   term to rewrite:  oldi*(oldi+1)
      axiom name[]:  multdistributeplus

       rewritebyaxiom multdistributeplus -- oldi * (oldi + 1)
                                          = oldi * oldi +
                                              oldi * 1
```

[12]The term t is of the form t^* if and only if $t = t^*[s_1/x_1, \ldots, s_n/x_n]$, where the s_i's are terms and the x_i's are variables of t^*.

```
<sdvs.2.1.2.9>  simp
   expression:  (oldi+1)*(oldi+2)=oldi*oldi+3*oldi+2

 true
```

The goal is now known to be true:

```
<sdvs.2.1.2.9>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.
```

We close the two proofs and "quit":

```
<sdvs.2.1.2.9>  close

     close -- 8 steps/applications

   join induction cases -- [sd pre: (1 le n)
                            comod: (all)
                              mod: (i,sigma)
                             post: (#i = n,
                                    2 * #sigma = n * (n + 1))]

 Complete the proof.

<sdvs.2.2>  close

 close -- 1 steps/applications

<sdvs.3>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.
```

6.2 Creating, Proving, and Invoking Lemmas

Example 11 In this example, we will prove *gauss.sd* once more, following the proof in the last example up to the point at which we invoked the command **rewritebyaxiom**. At that

point we will create a lemma, and, without proving it, we will use it to complete the proof. When we "quit" the proof of *gauss.sd*, SDVS will inform us that we used this lemma in the proof, but that a proof had not been provided (prior to the proof of *gauss.sd*).

We start the proof of *gauss.sd* by using the partial proof that we dumped in the previous example.

```
<sdvs.1>  init
   proof name[]:  gauss.partial1.proof

State Delta Verification System, Version 11

Restricted to authorized users only.

 setflag autoclose -- off

 open -- [sd pre: (covering(all,i,sigma),.i = 1,n ge 1,.sigma = 1,
                   formula(gaussloop.sd))
          comod: (all)
            mod: (i,sigma)
           post: (2 * #sigma = n * (n + 1),#i = n)]

   induction -- .i from 1 to n

     open -- [sd pre: (true)
              comod: (all)
               post: (2 * .sigma = .i * (.i + 1),.i = 1)]

     close -- 0 steps/applications

     open -- [sd pre: (.i ge 1,.i lt n,
                       2 * .sigma = .i * (.i + 1))
                mod: (i,sigma)
               post: (2 * #sigma = #i * (#i + 1),#i = .i + 1)]

       let -- oldi = .i

       let -- oldsigma = .sigma

       apply -- [sd pre: (.i lt n)
                  mod: (sigma,i)
                 post: (#i = .i + 1,#sigma = .sigma + #i)]

 Complete the proof.
```

We are now at the point of the proof at which we previously used the **rewritebyaxiom**

command. Instead of using the axioms, we "create" a lemma and use it instead. (We should point out that we could have created the lemma prior to the initiation of this proof.)

```
<sdvs.1>  createlemma
                  name:  gausslemma
               pattern:  (a+b)*(c+d)=c*a+d*a+c*b+d*b
      free variables[]:  a,b,c,d
    constant symbols[]:  <CR>
    function symbols[]:  <CR>
   predicate symbols[]:  <CR>

 Lemma 'gausslemma' created.
```

Note that the free variables a, b, c, and d that we used in the lemma are new. This is not required by SDVS, but since we will eventually have to prove the lemma "at the top level," (that is, not within another proof), we gain nothing by using variables in the current proof, since at the "top level" nothing will be known about these variables. The pattern of the lemma is a little odd, because we need this pattern only in the current proof, and because the more obvious equation $(a+b)*(c+d) = a*c+a*d+b*c+b*d$ requires four more invocations of the axiom "multcommute" in its proof.

We may now use the command **rewritebylemma**, which is entirely analogous to the **rewritebyaxiom** command, to complete the proof.

```
<sdvs.2.1.2.4>  rewritebylemma
   term to rewrite:  (oldi+0)*(oldi+1)
      lemma name[]:  gausslemma

       rewritebylemma gausslemma -- (oldi + 0) * (oldi + 1)
                                      = ((oldi * oldi + 1 * oldi) +
                                           oldi * 0) +
                                         1 * 0

<sdvs.2.1.2.5>  simp
   expression:  oldi*(oldi+1)=oldi*oldi+oldi

 true

<sdvs.2.1.2.5>  rewritebylemma
   term to rewrite:  (oldi+1)*(oldi+2)
      lemma name[]:  gausslemma

       rewritebylemma gausslemma -- (oldi + 1) * (oldi + 2)
                                      = ((oldi * oldi + 2 * oldi) +
                                           oldi * 1) +
                                         2 * 1
```

Note the way we entered the "term to rewrite" in the first invocation of the **rewrite-bylemma** command. We could not have entered the term $oldi*(oldi+1)$ instead, because it is not of the form of either of the two sides of the equality of the lemma, even though SDVS knows that $(oldi+0)$ is equivalent to *oldi*.

We may now close the proof as before, but we dump the proof prior to the "quit."

```
<sdvs.2.1.2.6>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.2.1.2.6>  close

     close -- 5 steps/applications

   join induction cases -- [sd pre: (1 le n)
                            comod: (all)
                              mod: (i,sigma)
                             post: (#i = n,
                                    2 * #sigma = n * (n + 1))]

 Complete the proof.

<sdvs.2.2>  close

 close -- 1 steps/applications

<sdvs.3>  dump-proof
   name:  gauss.partial2.proof

 Current proof dumped to gauss.partial2.proof.

<sdvs.3>  quit

 Proof session closed using unproved lemmas: (gausslemma)
 The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.
```

Note that SDVS has informed us that the unproved lemma *gausslemma* was used in the proof and that the customary "Q.E.D." did not follow the "quit".

Example 12 We will now prove the lemma that we created in the last example. First we will initialize the system and then use the SDVS command **provelemma** to open the proof. The lemma must be "created" prior to the invocation of this command. We are still in the session that we started at the beginning of this section. Thus SDVS knows that the lemma has been "created," and the current axioms include the axioms that we will need in its proof.

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  provelemma
   lemma name:  gausslemma
      proof[]:  <CR>

 open -- [sd pre: (true)
          post: ((a + b) * (c + d)
                   = ((c * a + d * a) + c * b) + d * b)]

<sdvs.1.1>  goals

 g(1) (a + b) * (c + d)
        = ((c * a + d * a) + c * b) + d * b
```

The **provelemma** command opens the proof of a state delta that has empty comodification and modification lists. The precondition is p if the lemma pattern is of the form $p \rightarrow q$ and is *true* otherwise. In the first case, the state delta asserts that henceforth (always) $p \rightarrow q$, and in the second case, the state delta asserts that henceforth (always) q is true.

We first **rewritebyaxiom** two times and then discuss another use of this command.

```
<sdvs.1.1>  rewritebyaxiom
   term to rewrite:  (a+b)*(c+d)
      axiom name[]:  multdistributeplus

   rewritebyaxiom multdistributeplus -- (a + b) * (c + d)
                                          = (a + b) * c +
                                              (a + b) * d

<sdvs.1.2>  rewritebyaxiom
   term to rewrite:  (a+b)*c
      axiom name[]:  multcommute
```

```
   rewritebyaxiom multcommute -- (a + b) * c = c * (a + b)
```

If an "axiom name" parameter is not given to the **rewritebyaxiom** command, SDVS searches the list of current axioms to find an axiom in which the "term to rewrite" parameter is of the form of one of the axiom equality terms. If it succeeds in this search, it rewrites the term parameter to the **rewritebyaxiom** command according to the axiom:

```
<sdvs.1.3>  rewritebyaxiom
   term to rewrite:  c*(a+b)
      axiom name[]:  <CR>

   rewritebyaxiom multdistributeplus -- c * (a + b)
                                          = c * a + c * b


<sdvs.1.4>  rewritebyaxiom
   term to rewrite:  (a+b)*d
      axiom name[]:  <CR>

   rewritebyaxiom multcommute -- (a + b) * d = d * (a + b)

<sdvs.1.5>  rewritebyaxiom
   term to rewrite:  d*(a+b)
      axiom name[]:  <CR>

   rewritebyaxiom multdistributeplus -- d * (a + b)
                                          = d * a + d * b


<sdvs.1.6>  close

 close -- 5 steps/applications

<sdvs.1>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  write
   path name[axioms/mult.axioms]:  tutorial/gauss.sdvs
         state delta names[]:  <CR>
               proof names[]:  <CR>
```

```
      axiom names[]:  multdistributeplus,multcommute
      lemma names[]:  gausslemma
    formula names[]:  <CR>
   formulas names[]:  <CR>
      macro names[]:  <CR>
   datatype names[]:  <CR>
   adalemma names[]:  <CR>

Write to file "tutorial/gauss.sdvs" -- (multdistributeplus,multcommute,
                                        gausslemma)
```

Once the proof of a lemma closes, SDVS automatically associates the proof with the lemma name. The lemma and its proof may be pretty-printed by the **pp** command:

```
<sdvs.2>  pp
   object:  lemma
   lemma name:  gausslemma

 lemma gausslemma (a,b,c,d): (a + b) * (c + d)
                               = ((c * a + d * a) + c * b) +
                                   d * b
<sdvs.2>  pp
   object:  lemmaproof
   lemma name:  gausslemma

 (provelemma gausslemma
    proof:
      (rewritebyaxiom (a + b) * (c + d)
         using: multdistributeplus,
       rewritebyaxiom (a + b) * c
         using: multcommute,
       rewritebyaxiom c * (a + b)
         using: multdistributeplus,
       rewritebyaxiom (a + b) * d
         using: multcommute,
       rewritebyaxiom d * (a + b)
         using: multdistributeplus,
       close))
```

If we were to use the **write** command and enter the name of the lemma to the query "lemma names," then SDVS would not only write the lemma to the specified file, but it would write its proof as well. For completeness, the names of the axioms used in the proof of the lemma should also be written to the file.

Thus, a **read** of the file would also read the proof of the lemma and the axioms used in its proof. The entire proof could then be run in the session in which this file was read.

The reader should use the **init** command with *gauss.partial2.proof* for the "proof name" parameter to see that after "quitting" the proof, SDVS will display "Q.E.D" and will no longer assert that unproved lemmas were used in the proof.

Example 13 Two other important static proof commands are **provebyaxiom** and **provebylemma**. The latter is used like the former once a lemma has been created. In this example we will create and prove a lemma using the **provebyaxiom** command.

The **provebyaxiom** command is used in a proof to assert an atomic formula q^* using an axiom either of the form $p \rightarrow q$ or the form q, where $q^* = q[t_1/x_1, \ldots, t_n/x_n]$, the t_i's are terms, the x_i's are variables of q, and where if the axiom is of the form $p \rightarrow q$, $p[t_1/x_1, \ldots, t_n/x_n]$ is known to be true in the current state.

We first create and open the proof of *square.ineq.lemma*, which asserts that if $b > a > 0$, then $b * b > a * a$. Since this lemma is an implication, the state delta that is created by the **provelemma** command has the antecedent in its precondition rather than *true*.

```
<sdvs.1>  createlemma
                    name:  square.ineq.lemma
                 pattern:  (0 lt a and a lt b) implies a*a lt b*b
       free variables[]:  a,b
     constant symbols[]:  <CR>
     function symbols[]:  <CR>
    predicate symbols[]:  <CR>

 Lemma 'square.ineq.lemma' created.

<sdvs.1>  init
    proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  provelemma
    lemma name:  square.ineq.lemma
       proof[]:  <CR>

 open -- [sd pre: (0 lt a & a lt b)
             post: (a * a lt b * b)]
```

```
<sdvs.1.1>  pp
   object:  axiom
   axiom name:  multgt

 axiom multgt (x,y,z):
          x gt 0 & y gt z or 0 gt x & z gt y --> x * y gt x * z
```

We use the axiom *multgt* in the **provebyaxiom** command. The substitutions, which are done automatically by SDVS, are b for x and y, and a for z. Note that the axiom is of the form $p \rightarrow q$ and that $p[b/x, b/y, a/z]$ is true.

```
<sdvs.1.1>  provebyaxiom
   formula to prove:  b*b gt b*a
       axiom name[]:  multgt

   provebyaxiom multgt -- b * b gt b * a

<sdvs.1.2>  provebyaxiom
   formula to prove:  a*b gt a*a
       axiom name[]:  <CR>

   provebyaxiom multgt -- a * b gt a * a
```

So far we have established that $b * b > b * a$ and that $a * b > a * a$. We must still prove that $b * a = a * b$. We do this, close and quit the proof, and pretty-print it.

```
<sdvs.1.3>  rewritebyaxiom
   term to rewrite:  a*b
      axiom name[]:  <CR>

   rewritebyaxiom multcommute -- a * b = b * a

<sdvs.1.4>  close

 close -- 3 steps/applications

<sdvs.1>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.
```

```
<sdvs.1>  pp
   object:  lemmaproof
   lemma name:  square.ineq.lemma

 (provelemma square.ineq.lemma
    proof:
      (provebyaxiom b * b gt b * a
        using: multgt,
       provebyaxiom a * b gt a * a
        using: multgt,
       rewritebyaxiom a * b
        using: multcommute,
       close))
```

If we were to prove that $b > a \geq 0 \rightarrow b * b > a * a$, we would have to do a proof by cases on the case $a = 0$.

7 Interaction with Application Languages

Our approach to adapting SDVS to the verification of programs in a new application language involves the following steps:

1. defining a sequence of application language subsets of increasing complexity, to be incorporated *incrementally* into SDVS;
2. defining in state deltas the semantics of the current application language subset; and
3. augmenting the SDVS Simplifier, domain repertoire, and proof rules with components necessary to support the application language subset.

As already indicated, an SDVS correctness proof of a subject program proceeds by symbolic execution of a state delta representation of the program, with the goal of achieving states in which the specification of the program holds. This state delta representation is obtained by invoking the application language translator, with the subject program as its argument.

The SDVS language translators are all organized according to the same scheme:

Parsing. A program is parsed according to the concrete syntax for the language subset, producing an *abstract syntax tree* for manipulation by the two subsequent translation phases. We use our own tools to specify and implement this process.

Phase 1: Static Checking. This first phase of semantic analysis detects "static" errors, such as items undeclared before their use, inappropriate types, and semantically ill-formed constructs. Provided that no errors occur in Phase 1, an environment for the second (and final) phase of the translator is produced.

Phase 2: State Delta Generation. This final phase of semantic analysis generates the state deltas that define the semantics and allow the symbolic execution of the program.

The formal specification of both phases of each translator is written in a *continuation-style denotational semantics* [12]. Space limitations constrain the following discussion to be simplified and sketchy; full details appear in various reports [13, 14, 15, 16, 17]. The translator is a Common Lisp program, whose behavior is largely specified by the mathematical equations of a continuation-style denotational semantics for the application language in terms of the state delta logic.

The implementation of the translator from the semantics is carried out automatically by a tool, also developed at The Aerospace Corporation, called the Denotational Semantics Translator Environment (DENOTE) [18]. The DENOTE Language (DL) was specifically designed for expressing semantic equations. DENOTE translates specifications written in DL, generating both (a) documentation in the form of formatted equations, and (b) a Common Lisp implementation of the equations. DENOTE is employed for the development of all language interfaces to SDVS.

Figure 6 shows SDVS's scheme for translating a computer program into the state delta language. Single vectors represent inputs; triple vectors represent the generation of programs, data or text; oval areas surround SDVS input; single-boxed areas surround text files, which are either output or data or both; and double-boxed areas represent programs. The areas enclosed in dashed boxes need be executed only once for each language – once for the grammar and once for the semantic definition of the translator.

Consider a computer language L being adapted to SDVS. First an SLR(1) grammar is created with a grammar analysis tool. This grammar is input to a parser generator tool whose output is a parser table. Any program written in L can be parsed by the table-driven parser along with the generated table for L. This parser is created once for the language and, when it is input a syntactically legal program, it outputs an abstract syntax tree for that program. The abstract syntax tree is input to the two phases of the translator backend. The backend is also generated once for the language and is created by the DENOTE tool. DENOTE accepts a set of equations that define the semantics of L, and outputs either the translator backend for L or the formatted equations for L.

A correctness proof of an application program must deal not only with the program's flow of control, but also with its data types. The language translator is responsible for properly modeling control flow, but knowledge about the data types must be incorporated into SDVS's inference mechanism (the Simplifier and domain axiomatizations). One must decide which existing components of SDVS can be directly adapted to deal with the new data types, and what enhancements to SDVS's inference machinery must be made.

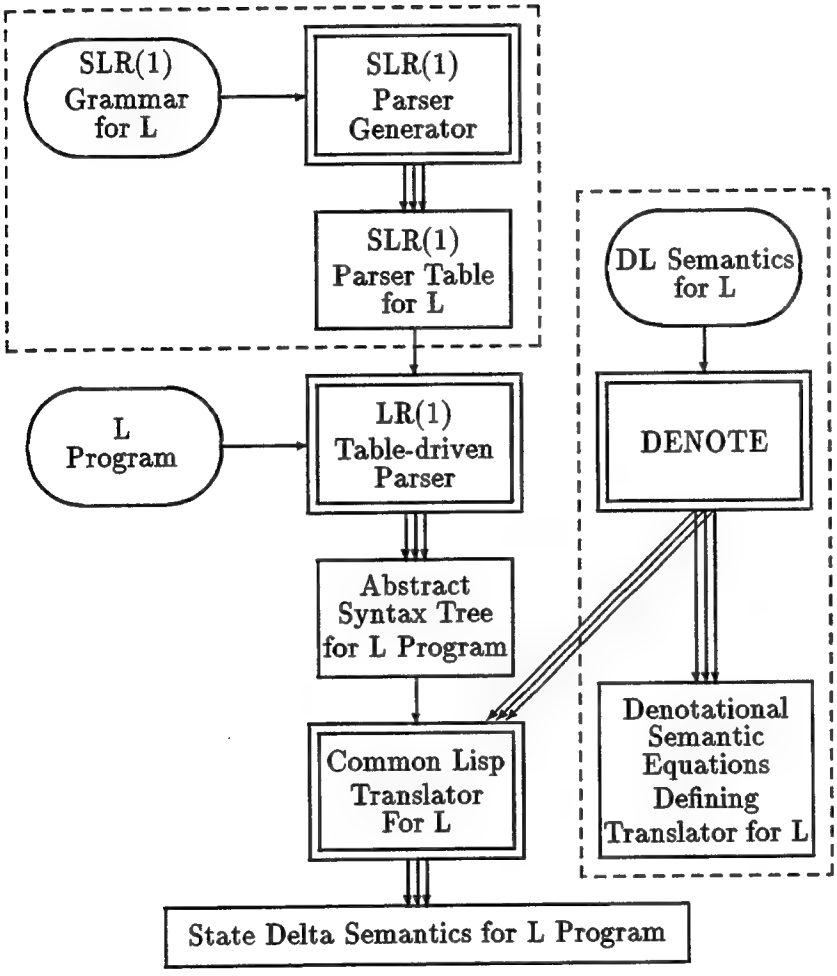

Figure 6: Parsing and Translating an L Program

7.1 Ada

The purpose of this section is to illustrate most of the techniques available to the user of SDVS in proofs of correctness properties of Ada programs.

The first three examples involve two versions of a simple Ada program with a procedure call. The main program orders two integers by calling a switch procedure, if the integers are not in the right order. The first version assigns concrete values to the integer variables, while the second version reads the (symbolic) values of the variables from a standard input file and, after ordering them, outputs these values to a standard output file. For the first version, we can only prove that the program terminates. We provide two proofs of this fact: the first proof introduces the salient features of an Ada proof, while the second illustrates the creation, proof, and invocation of an Ada lemma. In the third example, we use the second version of the program and prove a specification that relates the output to the input.

The last example is an Ada version of Example 8 in Section 3.4. The Ada program reads two integers, x and y from the standard input file, and if $y \geq 0$, computes their sum by means of a while loop and writes the sum to the standard output file. We include this example to illustrate the SDVS translation of an Ada while loop and the use of the **induct** command in its symbolic execution.

In SDVS (Version 11), a theorem concerning a correctness property of the Ada program **mainprogname** stored in the file **progfile.ada** is a state delta of the form

```
[sd pre: ada(progfile.ada), <input specifications>
  comod; all
    mod: all
   post: terminated(mainprogname), <input-output specifications>]
```

The formula `ada(progfile.ada)` is a translation of `mainprogname` into the language of the state delta logic. This formula acquires a meaning only after the invocation of the **adatr** command, at the top level, with the only argument to the command being the path name of the file `progfile.ada`. The predicate `terminated(mainprogname)` is asserted by SDVS at the last state of the symbolic execution of `ada(progfile.ada)`. The last two examples of this section illustrate the use of the input and the input-output specifications.

7.1.1 A simple Ada program with a subprogram

Example 14 Consider the following Ada program which is stored in the file `order1.ada`.

```
with text_io; use text_io;
with integer_io; use integer_io;
procedure order1 is
  u, v, switched : integer;
```

```
  procedure switch(x, y : in out integer) is
   temp : integer;
  begin
   temp:= x;
   x:= y;
   y:= temp;
  end switch;
begin
 u:= 3;
 v:= 2;
 if u <= v then
   switched :=0;
 else
   switch(u,v);
   switched := 1;
 end if;
end order1;
```

Clearly, the execution of this program terminates, and during the execution the values of the variables u and v are switched and the variable *switched* is assigned the value 1. Since these variables have neither initial nor final values, and in fact cannot be referenced at the top level of the precondition or postcondition of a state delta asserting a correctness property of the program (in the current SDVS semantic interpretation of Ada, these variables do not even exist at the initial and final states of the execution of the program), the only correctness property that we can prove about this program is that it terminates. The state delta *order1.sd* asserts this correctness property:

```
[sd pre: (ada(order1.ada))
  comod: (all)
    mod: (all)
   post: (terminated(order1))]
```

To prove this state delta we must first invoke the **adatr** command, at the top level, and then open the proof.

```
<sdvs.1>  adatr
   path name[testproofs/foo.ada]:  tutorial/order1.ada

 Parsing Stage 3 Ada file -- "tutorial/order1.ada"

 Translating Stage 3 Ada file -- "tutorial/order1.ada"

<sdvs.2>  setflag
```

```
   flag variable:  autoclose
   on or off[on]:  off

 setflag autoclose -- off

<sdvs.3>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  order1.sd
   proof[]:  <CR>

 open -- [sd pre: (ada(order1.ada))
          comod: (all)
            mod: (all)
           post: (terminated(order1))]

 Complete the proof.

<sdvs.1.1>  goals

 g(1) terminated(order1)
```

Here is the state delta that is usable at the beginning of the proof.

```
<sdvs.1.1>  usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc)
        post: (<adatr procedure order1 is
                        u, ... : integer
                        ...
                      begin
                        u := 3;
                        ...
                      end order1;>)]


 No usable quantified formulas.
```

The only element in the modification list of this usable and applicable state delta is the variable *order1\pc*, which represents the program counter of the main program `order1`. The program counter will appear in the modification list of every state delta that is generated by the Ada translator. We apply this state delta and check the next state delta that is generated by the translator.

```
<sdvs.1.1>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (<adatr procedure order1 is
                            u, ... : integer
                            ...
                          begin
                            u := 3;
                            ...
                          end order1;>)]
```

```
<sdvs.1.2>  usable

 u(1) [sd pre: (true)
        comod: (all)
          mod: (order1\pc,order1)
         post: (alldisjoint(order1,.order1,u,v,switched),
                covering(#order1,.order1,u,v,switched),
                declare(u,type(integer)),declare(v,type(integer)),
                declare(switched,type(integer)),
                <adatr u, ... : integer>)]


 No usable quantified formulas.
```

This usable state delta elaborates and declares the objects u, v, and *switched* to be of the type specified by the Ada program. We apply it and thereby elaborate the variables.

```
<sdvs.1.2>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1)
              post: (alldisjoint(order1,.order1,u,v,switched),
```

```
                    covering(#order1,.order1,u,v,switched),
                    declare(u,type(integer)),declare(v,type(integer)),
                    declare(switched,type(integer)),
                    <adatr u, ... : integer>)]
```

<sdvs.1.3> *usable*

```
u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc,u)
        post: (#u = 3,
               <adatr u := 3;>)]
```

```
No usable quantified formulas.
```

We first execute past the two assignment statements.

<sdvs.1.3> *apply*
sd/number[highest applicable/once]: *2*

```
apply -- [sd pre: (true)
           comod: (all)
             mod: (order1\pc,u)
            post: (#u = 3,
                   <adatr u := 3;>)]

apply -- [sd pre: (true)
           comod: (all)
             mod: (order1\pc,v)
            post: (#v = 2,
                   <adatr v := 2;>)]
```

<sdvs.1.5> *usable*

```
u(1) [sd pre: (~(.u le .v))
       comod: (all)
         mod: (order1\pc)
        post: (<adatr if u <= v
                        switched := 0;
                      else switch (u, ...);
                           ...
                      end if;>)]
```

```
u(2) [sd pre: (.u le .v)
       comod: (all)
         mod: (order1\pc)
        post: (<adatr if u <= v
                        switched := 0;
                      else switch (u, ...);
                           ...
                      end if;>)]
```

```
No usable quantified formulas.
```

<sdvs.1.5> *applicable*

```
u(1) [sd pre: (~(.u le .v))
       comod: (all)
         mod: (order1\pc)
        post: (<adatr if u <= v
                        switched := 0;
                      else switch (u, ...);
                           ...
                      end if;>)]
```

The conjunction of the two usable state deltas $u(1)$ and $u(2)$ is the SDVS translation of the "`if ... then ... else ... end if`" program segment. Since u is greater than v, only $u(1)$ is applicable.

<sdvs.1.5> *apply*

```
   sd/number[highest applicable/once]: <CR>

   apply -- [sd pre: (~(.u le .v))
              comod: (all)
                mod: (order1\pc)
               post: (<adatr if u <= v
                               switched := 0;
                             else switch (u, ...);
                                  ...
                             end if;>)]
```

<sdvs.1.6> *usable*

```
u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc,order1)
        post: (alldisjoint(order1,.order1,x,y),
```

```
                covering(#order1,.order1,x,y),declare(x,type(integer)),
                declare(y,type(integer)),
                <adatr switch (u, ...)>)]

 No usable quantified formulas.
```

This usable state delta, $u(1)$, represents the elaboration of the objects x and y of procedure switch. We elaborate the declarations, bind x and y to u and v, respectively, and advance the program counter to "at(order1.switch)".

```
<sdvs.1.6>  apply
   sd/number[highest applicable/once]:  3

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1)
              post: (alldisjoint(order1,.order1,x,y),
                     covering(#order1,.order1,x,y),
                     declare(x,type(integer)),declare(y,type(integer)),
                     <adatr switch (u, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,x,y)
              post: (#x = .u,#y = .v,
                     <adatr switch (u, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (#order1\pc = at(order1.switch),
                     <adatr switch (u, ...)>)]

<sdvs.1.9>  usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc,order1)
        post: (alldisjoint(order1,.order1,temp),
               covering(#order1,.order1,temp),
               declare(temp,type(integer)),
               <adatr temp : integer>)]
```

```
No usable quantified formulas.
```

The identifier `order1.switch` is the *qualified name* of procedure `switch`, and the usable state delta $u(1)$ represents the elaboration of the object *temp* of procedure `switch`. We elaborate the declaration, and execute through the body of the procedure.

```
<sdvs.1.9>  apply
   sd/number[highest applicable/once]:  4

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1)
              post: (alldisjoint(order1,.order1,temp),
                     covering(#order1,.order1,temp),
                     declare(temp,type(integer)),
                     <adatr temp : integer>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,temp)
              post: (#temp = .x,
                     <adatr temp := x;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,x)
              post: (#x = .y,
                     <adatr x := y;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,y)
              post: (#y = .temp,
                     <adatr y := temp;>)]


<sdvs.1.13>  simp
   expression:  .x

 2

<sdvs.1.13>  simp
   expression:  .y

 3
```

```
<sdvs.1.13>  applicable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc,order1,temp)
        post: (covering(.order1,#order1,temp),undeclare(temp),
               <adatr temp : integer>)]
```

We now "undeclare" the object *temp* and advance the state to the point at which the program counter is at "exited(order1.switch)".

```
<sdvs.1.13>  apply
   sd/number[highest applicable/once]:  2

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1,temp)
              post: (covering(.order1,#order1,temp),undeclare(temp),
                     <adatr temp : integer>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (#order1\pc = exited(order1.switch),
                     <adatr switch (u, ...)>)]

<sdvs.1.15>  simp
   expression:  .x

 2

<sdvs.1.15>  simp
   expression:  .y

 3
```

Notice that, at this point, x and y have not been "undeclared." We assign the current values of x and y to u and v, respectively, and "undeclare" x and y.

```
<sdvs.1.15>  apply
   sd/number[highest applicable/once]:  2

   apply -- [sd pre: (true)
             comod: (all)
```

```
                mod: (order1\pc,u,v)
               post: (#u = .x,#v = .y,
                      <adatr switch (u, ...)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order1\pc,order1,x,y)
             post: (covering(.order1,#order1,x,y),undeclare(x,y),
                    <adatr switch (u, ...)>)]
```

<sdvs.1.17> *usable*

```
u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc,switched)
        post: (#switched = 1,
               <adatr switched := 1;>)]


No usable quantified formulas.
```

We have now truly exited the procedure `switch` and are at the assignment `switched := 1` of the main program. We execute the assignment, undeclare u and v, and close, quit, and pretty-print the proof.

<sdvs.1.17> *apply*
sd/number[highest applicable/once]: *<CR>*

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (order1\pc,switched)
           post: (#switched = 1,
                  <adatr switched := 1;>)]
```

<sdvs.1.18> *apply*
sd/number[highest applicable/once]: *2*

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (order1\pc,order1,u,v,switched)
           post: (covering(.order1,#order1,u,v,switched),
                  undeclare(u,v,switched),
                  <adatr u, ... : integer>)]
```

```
   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (terminated(order1))]

<sdvs.1.20>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.1.20>  close

 close -- 19 steps/applications

<sdvs.2>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  prove order1.sd
    proof:
      (apply u(1),
       apply u(1),
       apply 2,
       apply u(1),
       apply 11,
       apply u(1),
       apply 2,
       close)
```

The **apply**s above could be replaced by one use of the **go** command.

7.1.2 Creating, proving, and invoking an Ada lemma

Example 15 In this example we create and prove a lemma about the subprogram **switch** of the program **order1**, and then reprove the state delta *order1.sd* of Example 14 by invoking the lemma at the appropriate point in the proof. We also demonstrate the **go** command in the elaboration of the declarations.

We first initialize the system and translate the Ada program in the file **order1.ada**. (It is not necessary to translate the file again if it has been translated in the current session.)

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.
```

We can now create the Ada lemma.

```
<sdvs.1>  createadalemma
        lemma name:  switch.lemma
         file name:  tutorial/order1.ada
   subprogram name:  switch
    qualified name:  order1.switch
   preconditions[]:  <CR>
        mod list[]:  x,y
    postconditions:  #x=.y,#y=.x

 createadalemma -- [sd pre: (.order1\pc = at(order1.switch))
                    comod: (all)
                      mod: (order1\pc,x,y)
                     post: (#x = .y,#y = .x,
                            #order1\pc = exited(order1.switch))]
```

Notice that the system created a state delta with some of our entries and supplied additional ones. Specifically, the precondition and postcondition include the expected values of the program counter of the main program **order1**, in which the subprogram **switch** is contained. In Example 14 we saw at which points of the execution these values were attained. The precondition value of the program counter was attained after the declaration and binding of the variables x and y, and the postcondition value was attained after the local variable *temp* was undeclared. Note that we entered a null parameter to the **preconditions[]:** query of the **createadalemma** command. The user may enter any precondition formula whose program variables are either in the scope of the subprogram declaration or are formal parameters of the subprogram. Of course, this formula must be true at the time that the lemma is invoked. The program counter was also added to the modification list of the state delta. Finally, this is the state delta that will be asserted by the **invokeadalemma** command.

We will now prove this Ada lemma. The **proveadalemma** command sets up the environment to prove the lemma.

```
<sdvs.2>  setflag
   flag variable:  autoclose
   on or off[on]:  on

 setflag autoclose -- on

<sdvs.3>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  proveadalemma
    Ada lemma name:  switch.lemma
           proof[]:  <CR>

 open -- [sd pre: (alldisjoint(order1,.order1),
                   covering(.order1,order1\pc,u,v,switched,stdin,stdin\ctr,
                            stdout,stdout\ctr),
                   declare(stdout\ctr,type(integer)),
                   declare(stdout,type(polymorphic)),
                   declare(stdin\ctr,type(integer)),
                   declare(stdin,type(polymorphic)),
                   declare(switched,type(integer)),
                   declare(v,type(integer)),declare(u,type(integer)),
                   <adatr switch (x, ...);>)
           comod: (all)
             mod: (all)
            post: ([sd pre: (.order1\pc = at(order1.switch))
                      comod: (all)
                        mod: (diff(all,
                                   diff(union(order1\pc,u,v,switched,stdin,
                                              stdin\ctr,stdout,stdout\ctr,x,
                                              y),
                                        union(order1\pc,x,y))))
                       post: (#x = .y,#y = .x,
                              #order1\pc = exited(order1.switch))])]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order1\pc,order1)
```

```
              post: (alldisjoint(order1,.order1,x,y),
                     covering(#order1,.order1,x,y),
                     declare(x,type(integer)),declare(y,type(integer)),
                     <adatr null;>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order1\pc,x,y)
             post: (#x = .x,#y = .y,
                    <adatr null;>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order1\pc)
             post: (#order1\pc = at(order1.switch),
                    <adatr null;>)]

  go -- breakpoint reached

  open -- [sd pre: (.order1\pc = at(order1.switch))
           comod: (all)
             mod: (diff(all,
                        diff(union(order1\pc,u,v,switched,stdin,
                                   stdin\ctr,stdout,stdout\ctr,x,y),
                             union(order1\pc,x,y))))
            post: (#x = .y,#y = .x,
                   #order1\pc = exited(order1.switch))]
```

<sdvs.1.4.1> *usable*

```
u(1) [sd pre: (true)
      comod: (all)
        mod: (order1\pc,order1)
       post: (alldisjoint(order1,.order1,temp),
              covering(#order1,.order1,temp),
              declare(temp,type(integer)),
              <adatr temp : integer>)]


No usable quantified formulas.
```

The proof proceeds as before; we use the **until** command for the applications.

<sdvs.1.4.1> *until*
formula: *#order1\pc=exited(order1.switch)*

```
    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc,order1)
               post: (alldisjoint(order1,.order1,temp),
                      covering(#order1,.order1,temp),
                      declare(temp,type(integer)),
                      <adatr temp : integer>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc,temp)
               post: (#temp = .x,
                      <adatr temp := x;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc,x)
               post: (#x = .y,
                      <adatr x := y;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc,y)
               post: (#y = .temp,
                      <adatr y := temp;>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc,order1,temp)
               post: (covering(.order1,#order1,temp),undeclare(temp),
                      <adatr temp : integer>)]

    apply -- [sd pre: (true)
              comod: (all)
                mod: (order1\pc)
               post: (#order1\pc = exited(order1.switch),
                      <adatr null;>)]

  close -- 6 steps/applications

close -- 4 steps/applications

proveadalemma -- [sd pre: (.order1\pc = at(order1.switch))
                  comod: (all)
```

```
                    mod: (order1\pc,x,y)
                   post: (#x = .y,#y = .x,
                          #order1\pc = exited(order1.switch))]

<sdvs.1>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  lemmaproof
   lemma name:  switch.lemma

 ()
```

Let us once more open the proof of *order*1.*sd*.

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  order1.sd
   proof[]:  <CR>

 open -- [sd pre: (ada(order1.ada))
          comod: (all)
            mod: (all)
           post: (terminated(order1))]

 Complete the proof.
```

The **go** command is similar to **until** except that

- in the application of state deltas, **go** will only apply a state delta if it is applicable *and* at the top of the usable state deltas stack,
- **until** requires a formula parameter whereas **go** does not, and

- **go** also instantiates quantified formulas that are "applicable."

```
<sdvs.1.1>  go
   until[]:  #order1\pc=at(order1.switch)

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (<adatr procedure order1 is
                          u, ... : integer
                          ...
                        begin
                          u := 3;
                          ...
                        end order1;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1)
              post: (alldisjoint(order1,.order1,u,v,switched),
                     covering(#order1,.order1,u,v,switched),
                     declare(u,type(integer)),declare(v,type(integer)),
                     declare(switched,type(integer)),
                     <adatr u, ... : integer>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,u)
              post: (#u = 3,
                     <adatr u := 3;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,v)
              post: (#v = 2,
                     <adatr v := 2;>)]

   apply -- [sd pre: (~(.u le .v))
             comod: (all)
               mod: (order1\pc)
              post: (<adatr if u <= v
                            switched := 0;
                          else switch (u, ...);
                               ...
                          end if;>)]
```

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (order1\pc,order1)
           post: (alldisjoint(order1,.order1,x,y),
                  covering(#order1,.order1,x,y),
                  declare(x,type(integer)),declare(y,type(integer)),
                  <adatr switch (u, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order1\pc,x,y)
           post: (#x = .u,#y = .v,
                  <adatr switch (u, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order1\pc)
           post: (#order1\pc = at(order1.switch),
                  <adatr switch (u, ...)>)]

go -- breakpoint reached
```

<sdvs.1.9> *usable*

```
u(1) [sd pre: (true)
      comod: (all)
        mod: (order1\pc,order1)
       post: (alldisjoint(order1,.order1,temp),
              covering(#order1,.order1,temp),
              declare(temp,type(integer)),
              <adatr temp : integer>)]


No usable quantified formulas.
```

We are at the point at which the precondition of the state delta created by the **createadalemma** command is true, and we may apply the lemma using the **invokeadalemma** command. This is the only point where we may use the lemma, because it is the only point at which the program counter has the correct value.

<sdvs.1.9> *invokeadalemma*

```
Ada lemma name: switch.lemma

invokeadalemma -- [sd pre: (.order1\pc = at(order1.switch))
```

```
                    comod: (all)
                      mod: (order1\pc,x,y)
                     post: (#x = .y,#y = .x,
                            #order1\pc = exited(order1.switch),
                            <adatr return;>)]

<sdvs.1.10>  usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (order1\pc)
        post: (#order1\pc = exited(order1.switch),
               <adatr switch (u, ...)>)]


 No usable quantified formulas.

<sdvs.1.10>  apply
   sd/number[highest applicable/once]: <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (#order1\pc = exited(order1.switch),
                     <adatr switch (u, ...)>)]

<sdvs.1.11>  simp
   expression: .x

 2

<sdvs.1.11>  simp
   expression: .y

 3

<sdvs.1.11>  simp
   expression: .u

 3

<sdvs.1.11>  simp
   expression: .v

 2
```

After the application of the state delta created by the lemma, we are at a familiar point in the proof; we assign the values of x and y to u and v and execute to the end using **until**. Finally, we close and quit the proof, then pretty-print it.

```
<sdvs.1.11>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,u,v)
              post: (#u = .x,#v = .y,
                     <adatr switch (u, ...)>)]

<sdvs.1.12>  simp
   expression:  .u

 2

<sdvs.1.12>  simp
   expression:  .v

 3

<sdvs.1.12>  until
   formula:  terminated(order1)

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1,x,y)
              post: (covering(.order1,#order1,x,y),undeclare(x,y),
                     <adatr switch (u, ...)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,switched)
              post: (#switched = 1,
                     <adatr switched := 1;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc,order1,u,v,switched)
              post: (covering(.order1,#order1,u,v,switched),
                     undeclare(u,v,switched),
                     <adatr u, ... : integer>)]
```

```
   apply -- [sd pre: (true)
             comod: (all)
               mod: (order1\pc)
              post: (terminated(order1))]

 close -- 15 steps/applications

<sdvs.2>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  prove order1.sd
    proof:
      (go #order1\pc = at(order1.switch),
       invokeadalemma switch.lemma,
       apply u(1),
       apply u(1),
       until terminated(order1))
```

7.1.3 Ada input and output

The program order1 in Examples 14 and 15 did not read or write any values of its objects. Consequently, the only specification we could state and prove about the program was that it terminated. In practice, a program inputs and outputs data, and a specification concerning it is usually a relation of the output to the input.

Standard input and output buffers are part of the predefined environment for SDVS Ada programs. The SDVS Ada translator behaves as if every main program contains a package roughly of the form

```
package STANDARD is
 package TEXT_IO is
   stdin      : array(1..?) of polymorphic;
```

```
stdin\ctr  : integer := 1;
stdout     : array(1..?) of polymorphic;
stdout\ctr : integer := 1;
procedure get(get\item : out polymorphic) is
 begin
  get\item  := stdin(stdin\ctr);
  stdin\ctr := stdin\ctr+1;
 end get;
procedure put(put\item : in polymorphic) is
 begin
  stdout(stdout\ctr) := put\item;
  stdout\ctr := stdout\ctr+1;
 end put;
```

The reader may have noticed in the last example that the **proveadalemma** command opened the proof of a state delta, two of whose fields allude to the objects in this fictional "STANDARD" package. A "get(u)" or a "put(u)" in an Ada program is translated as if it were a procedure call to the get and put procedures of this package.

Our next example concerns an Ada program with both input and output and a state delta assertion of the correctness of its output with respect to its input.

Example 16 Consider the program order2 in the file order2.ada:

```
with text_io; use text_io;
with integer_io; use integer_io;

procedure order2 is
 u, v, switched : integer;
  procedure switch(x, y : in out integer) is
   temp : integer;
  begin
   temp:= x;
   x:= y;
   y:= temp;
  end switch;
begin
 get(u);
 get(v);
 if u <= v then
   switched :=0;
 else
   switch(u,v);
   switched := 1;
```

```
 end if;
 put(u);
 put(v);
 put(switched);
end order2;
```

and the state delta *order2.sd*:

```
  [sd pre: (ada(order2.ada))
    comod: (all)
      mod: (all)
     post: (terminated(order2),#stdout[1] le #stdout[2],
            (#stdout[3] = 0 & #stdout[1] = .stdin[1]) &
               #stdout[2] = .stdin[2] or
              (#stdout[3] = 1 & #stdout[1] = .stdin[2]) &
                #stdout[2] = .stdin[1])]
```

In the symbolic execution of `ada(order2.ada)`, i.e., of the SDVS translation of the program `order2`, u and v will be initially assigned the values of *stdin*[1] and *stdin*[2], respectively, and towards the end of the execution, *stdout*[1], *stdout*[2], and *stdout*[3] will be assigned the values of u, v, and *switched*, respectively. It should be obvious from the program that *order2.sd* is a valid state delta (provided that *stdin*[1] and *stdin*[2] remain constant during the execution).

Here is a proof of *order2.sd*.

```
<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[off]:  on

 setflag autoclose -- on

<sdvs.2>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  adatr
   path name[tutorial/order1.ada]:  tutorial/order2.ada

 Parsing Stage 3 Ada file -- "tutorial/order2.ada"
```

```
Translating Stage 3 Ada file -- "tutorial/order2.ada"

<sdvs.2>  pp
   object:  ada
   file name[order2.ada]:  order2.ada

 alldisjoint(order2,.order2)
 covering(.order2,order2\pc,stdin,stdin\ctr,stdout,stdout\ctr)
 declare(stdin,type(array,1,range(stdin),type(polymorphic)))
 declare(stdin\ctr,type(integer))
 .stdin\ctr = 1
 declare(stdout,type(array,1,range(stdout),type(polymorphic)))
 declare(stdout\ctr,type(integer))
 .stdout\ctr = 1


<sdvs.2>  prove
   state delta[]:  order2.sd
   proof[]:  <CR>

 open -- [sd pre: (ada(order2.ada))
           comod: (all)
             mod: (all)
            post: (terminated(order2),#stdout[1] le #stdout[2],
                   (#stdout[3] = 0 & #stdout[1] = .stdin[1]) &
                      #stdout[2] = .stdin[2] or
                     (#stdout[3] = 1 & #stdout[1] = .stdin[2]) &
                       #stdout[2] = .stdin[1])]

 Complete the proof.

<sdvs.2.1>  usable

 u(1) [sd pre: (true)
         comod: (all)
           mod: (order2\pc)
          post: (<adatr procedure order2 is
                              u, ... : integer
                              ...
                            begin
                              get (u);
                              ...
                            end order2;>)]
```

```
No usable quantified formulas.
```

Notice that it is possible to pretty-print (a portion of) the translation of the Ada program, and that the parameter given to the **pp** command is the file name of the Ada program, not the name of the program itself.

We now proceed as before.

```
<sdvs.2.1>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc)
              post: (<adatr procedure order2 is
                           u, ... : integer
                           ...
                         begin
                           get (u);
                           ...
                         end order2;>)]

<sdvs.2.2>  apply
   sd/number[highest applicable/once]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc,order2)
              post: (alldisjoint(order2,.order2,u,v,switched),
                     covering(#order2,.order2,u,v,switched),
                     declare(u,type(integer)),declare(v,type(integer)),
                     declare(switched,type(integer)),
                     <adatr u, ... : integer>)]

<sdvs.2.3>  usable

 u(1) [sd pre: (true)
       comod: (all)
         mod: (order2\pc,order2)
        post: (alldisjoint(order2,.order2,get\item),
               covering(#order2,.order2,get\item),
               declare(get\item,type(polymorphic)),
               <adatr get (u)>)]


No usable quantified formulas.
```

The applicable state delta $u(1)$ is the beginning of the translation of the get(u) statement. We execute through its full translation.

```
<sdvs.2.3>  apply
   sd/number[highest applicable/once]:  6

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc,order2)
              post: (alldisjoint(order2,.order2,get\item),
                     covering(#order2,.order2,get\item),
                     declare(get\item,type(polymorphic)),
                     <adatr get (u)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc)
              post: (#order2\pc = at(standard.text_io.get),
                     <adatr get (u)>)]

   apply -- [sd pre: (.order2\pc = at(standard.text_io.get))
             comod: (all)
               mod: (order2\pc,stdin\ctr,get\item)
              post: (#get\item = .stdin[.stdin\ctr],
                     #stdin\ctr = .stdin\ctr + 1,
                     #order2\pc = exited(standard.text_io.get),
                     <adatr null;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc)
              post: (#order2\pc = exited(standard.text_io.get),
                     <adatr get (u)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc,u)
              post: (#u = .get\item,
                     <adatr get (u)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc,order2,get\item)
              post: (covering(.order2,#order2,get\item),
                     undeclare(get\item),
```

```
                    <adatr get (u)>)]
```

<sdvs.2.9> *usable*

```
 u(1) [sd pre: (true)
       comod: (all)
         mod: (order2\pc,order2)
        post: (alldisjoint(order2,.order2,get\item!2),
               covering(#order2,.order2,get\item!2),
               declare(get\item!2,type(polymorphic)),
               <adatr get (v)>)]


 No usable quantified formulas.
```

We are now at the beginning of the translation of the get(v) statement of the program. Having already seen the flow of the first get, we quickly proceed to the "if u <= v then ... else ... end if" statement, using go, which will stop at that point because there will be no applicable state deltas at the top of the usable stack to apply.

<sdvs.2.9> *go*
until[]: *<CR>*

```
   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc,order2)
              post: (alldisjoint(order2,.order2,get\item!2),
                     covering(#order2,.order2,get\item!2),
                     declare(get\item!2,type(polymorphic)),
                     <adatr get (v)>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (order2\pc)
              post: (#order2\pc = at(standard.text_io.get),
                     <adatr get (v)>)]

   apply -- [sd pre: (.order2\pc = at(standard.text_io.get))
             comod: (all)
               mod: (order2\pc,stdin\ctr,get\item!2)
              post: (#get\item!2 = .stdin[.stdin\ctr],
                     #stdin\ctr = .stdin\ctr + 1,
                     #order2\pc = exited(standard.text_io.get),
                     <adatr null;>)]
```

```
  apply -- [sd pre: (true)
            comod: (all)
              mod: (order2\pc)
             post: (#order2\pc = exited(standard.text_io.get),
                    <adatr get (v)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order2\pc,v)
             post: (#v = .get\item!2,
                    <adatr get (v)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (order2\pc,order2,get\item!2)
             post: (covering(.order2,#order2,get\item!2),
                    undeclare(get\item!2),
                    <adatr get (v)>)]

  go -- no more declarations or statements
```

<sdvs.2.15> *usable*

```
u(1) [sd pre: (~(.u le .v))
       comod: (all)
         mod: (order2\pc)
        post: (<adatr if u <= v
                        switched := 0;
                      else switch (u, ...);
                           ...
                      end if;>)]

u(2) [sd pre: (.u le .v)
       comod: (all)
         mod: (order2\pc)
        post: (<adatr if u <= v
                        switched := 0;
                      else switch (u, ...);
                           ...
                      end if;>)]


No usable quantified formulas.
```

<sdvs.2.15> *applicable*

The reason that neither of the two usable state deltas is applicable is that neither precondition is necessarily true. We must use the **cases** command to consider each possibility.

```
<sdvs.2.15>  cases
   case predicate:  .u le .v

   cases -- .u le .v

     open -- [sd pre: (.u le .v)
              comod: (all)
                mod: (all)
               post: (terminated(order2),#stdout[1] le #stdout[2],
                      (#stdout[3] = 0 & #stdout[1] = stdin\237) &
                         #stdout[2] = stdin\239 or
                        (#stdout[3] = 1 & #stdout[1] = stdin\239) &
                          #stdout[2] = stdin\237)]
```

The proof of the first case, in which $u \leq v$, has been opened by the system. The goal remains the same. We must execute until we reach the goal, at which point SDVS will open the proof of the second case. We execute to the put(u) statement.

```
<sdvs.2.15.1.1>  apply
   sd/number[highest applicable/once]:  2

      apply -- [sd pre: (.u le .v)
                comod: (all)
                  mod: (order2\pc)
                 post: (<adatr if u <= v
                                  switched := 0;
                               else switch (u, ...);
                                  ...
                             end if;>)]

      apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc,switched)
                 post: (#switched = 0,
                        <adatr switched := 0;>)]

<sdvs.2.15.1.3>  usable

 u(1) [sd pre: (true)
        comod: (all)
          mod: (order2\pc,order2)
         post: (alldisjoint(order2,.order2,put\item),
```

```
            covering(#order2,.order2,put\item),
            declare(put\item,type(polymorphic)),
            <adatr put (u)>)]

 No usable quantified formulas.
```

The put(u) statement is translated in a manner roughly akin to a procedure call to put. We now proceed through six applications to the next statement of the program order2.

```
<sdvs.2.15.1.3>  apply
   sd/number[highest applicable/once]:  6

      apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc,order2)
                 post: (alldisjoint(order2,.order2,put\item),
                        covering(#order2,.order2,put\item),
                        declare(put\item,type(polymorphic)),
                        <adatr put (u)>)]

      apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc,put\item)
                 post: (#put\item = .u,
                        <adatr put (u)>)]

      apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc)
                 post: (#order2\pc = at(standard.text_io.put),
                        <adatr put (u)>)]

      apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
                comod: (all)
                  mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
                 post: (#stdout[.stdout\ctr] = .put\item,
                        #stdout\ctr = .stdout\ctr + 1,
                        #order2\pc = exited(standard.text_io.put),
                        <adatr null;>)]

      apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc)
                 post: (#order2\pc = exited(standard.text_io.put),
```

```
                        <adatr put (u)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,order2,put\item)
                  post: (covering(.order2,#order2,put\item),
                         undeclare(put\item),
                         <adatr put (u)>)]
```

<sdvs.2.15.1.9> *usable*

```
 u(1) [sd pre: (true)
        comod: (all)
          mod: (order2\pc,order2)
         post: (alldisjoint(order2,.order2,put\item!2),
                covering(#order2,.order2,put\item!2),
                declare(put\item!2,type(polymorphic)),
                <adatr put (v)>)]


 No usable quantified formulas.
```

We are now at the beginning of the next two statements, put(v) and put(switched). We execute through both, to the end of the first case, using the **go** command.

<sdvs.2.15.1.9> *go*
until[]: *~(#u le #v)*

```
       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,order2)
                  post: (alldisjoint(order2,.order2,put\item!2),
                         covering(#order2,.order2,put\item!2),
                         declare(put\item!2,type(polymorphic)),
                         <adatr put (v)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,put\item!2)
                  post: (#put\item!2 = .v,
                         <adatr put (v)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc)
```

```
            post: (#order2\pc = at(standard.text_io.put),
                   <adatr put (v)>)]

apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
          comod: (all)
            mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
           post: (#stdout[.stdout\ctr] = .put\item!2,
                  #stdout\ctr = .stdout\ctr + 1,
                  #order2\pc = exited(standard.text_io.put),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = exited(standard.text_io.put),
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,put\item!2)
           post: (covering(.order2,#order2,put\item!2),
                  undeclare(put\item!2),
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2)
           post: (alldisjoint(order2,.order2,put\item!3),
                  covering(#order2,.order2,put\item!3),
                  declare(put\item!3,type(polymorphic)),
                  <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,put\item!3)
           post: (#put\item!3 = .switched,
                  <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = at(standard.text_io.put),
                  <adatr put (switched)>)]

apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
```

```
                comod: (all)
                  mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
                 post: (#stdout[.stdout\ctr] = .put\item!3,
                        #stdout\ctr = .stdout\ctr + 1,
                        #order2\pc = exited(standard.text_io.put),
                        <adatr null;>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc)
                 post: (#order2\pc = exited(standard.text_io.put),
                        <adatr put (switched)>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc,order2,put\item!3)
                 post: (covering(.order2,#order2,put\item!3),
                        undeclare(put\item!3),
                        <adatr put (switched)>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc,order2,u,v,switched)
                 post: (covering(.order2,#order2,u,v,switched),
                        undeclare(u,v,switched),
                        <adatr u, ... : integer>)]

    apply -- [sd pre: (true)
                comod: (all)
                  mod: (order2\pc)
                 post: (terminated(order2))]

close -- 22 steps/applications

open -- [sd pre: (~(.u le .v))
           comod: (all)
             mod: (all)
            post: (terminated(order2),#stdout[1] le #stdout[2],
                   (#stdout[3] = 0 & #stdout[1] = stdin\237) &
                      #stdout[2] = stdin\239 or
                     (#stdout[3] = 1 & #stdout[1] = stdin\239) &
                       #stdout[2] = stdin\237)]
```

Complete the proof.

The system has opened the proof of the second case ($u > v$). Since this case is similar to the first, we use **go** to reach one of our goals, `terminated(order2)`. Once this goal is reached, the other goals will have been achieved as well. No other commands are necessary for this simple proof.

```
<sdvs.2.15.2.1>  go
   until[]:  terminated(order2)

       apply -- [sd pre: (~(.u le .v))
                 comod: (all)
                   mod: (order2\pc)
                  post: (<adatr if u <= v
                                  switched := 0;
                                else switch (u, ...);
                                  ...
                                end if;>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,order2)
                  post: (alldisjoint(order2,.order2,x,y),
                         covering(#order2,.order2,x,y),
                         declare(x,type(integer)),
                         declare(y,type(integer)),
                         <adatr switch (u, ...)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,x,y)
                  post: (#x = .u,#y = .v,
                         <adatr switch (u, ...)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc)
                  post: (#order2\pc = at(order2.switch),
                         <adatr switch (u, ...)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (order2\pc,order2)
                  post: (alldisjoint(order2,.order2,temp),
                         covering(#order2,.order2,temp),
                         declare(temp,type(integer)),
                         <adatr temp : integer>)]
```

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,temp)
           post: (#temp = .x,
                  <adatr temp := x;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,x)
           post: (#x = .y,
                  <adatr x := y;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,y)
           post: (#y = .temp,
                  <adatr y := temp;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,temp)
           post: (covering(.order2,#order2,temp),undeclare(temp),
                  <adatr temp : integer>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = exited(order2.switch),
                  <adatr switch (u, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,u,v)
           post: (#u = .x,#v = .y,
                  <adatr switch (u, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,x,y)
           post: (covering(.order2,#order2,x,y),undeclare(x,y),
                  <adatr switch (u, ...)>)]

apply -- [sd pre: (true)
          comod: (all)
```

```
            mod: (order2\pc,switched)
           post: (#switched = 1,
                  <adatr switched := 1;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2)
           post: (alldisjoint(order2,.order2,put\item),
                  covering(#order2,.order2,put\item),
                  declare(put\item,type(polymorphic)),
                  <adatr put (u)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,put\item)
           post: (#put\item = .u,
                  <adatr put (u)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = at(standard.text_io.put),
                  <adatr put (u)>)]

apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
          comod: (all)
            mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
           post: (#stdout[.stdout\ctr] = .put\item,
                  #stdout\ctr = .stdout\ctr + 1,
                  #order2\pc = exited(standard.text_io.put),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = exited(standard.text_io.put),
                  <adatr put (u)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,put\item)
           post: (covering(.order2,#order2,put\item),
                  undeclare(put\item),
                  <adatr put (u)>)]
```

```
apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2)
           post: (alldisjoint(order2,.order2,put\item!2),
                  covering(#order2,.order2,put\item!2),
                  declare(put\item!2,type(polymorphic)),
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,put\item!2)
           post: (#put\item!2 = .v,
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = at(standard.text_io.put),
                  <adatr put (v)>)]

apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
          comod: (all)
            mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
           post: (#stdout[.stdout\ctr] = .put\item!2,
                  #stdout\ctr = .stdout\ctr + 1,
                  #order2\pc = exited(standard.text_io.put),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = exited(standard.text_io.put),
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,put\item!2)
           post: (covering(.order2,#order2,put\item!2),
                  undeclare(put\item!2),
                  <adatr put (v)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2)
           post: (alldisjoint(order2,.order2,put\item!3),
```

```
                    covering(#order2,.order2,put\item!3),
                    declare(put\item!3,type(polymorphic)),
                    <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,put\item!3)
           post: (#put\item!3 = .switched,
                  <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = at(standard.text_io.put),
                  <adatr put (switched)>)]

apply -- [sd pre: (.order2\pc = at(standard.text_io.put))
          comod: (all)
            mod: (order2\pc,stdout[.stdout\ctr],stdout\ctr)
           post: (#stdout[.stdout\ctr] = .put\item!3,
                  #stdout\ctr = .stdout\ctr + 1,
                  #order2\pc = exited(standard.text_io.put),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc)
           post: (#order2\pc = exited(standard.text_io.put),
                  <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,put\item!3)
           post: (covering(.order2,#order2,put\item!3),
                  undeclare(put\item!3),
                  <adatr put (switched)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (order2\pc,order2,u,v,switched)
           post: (covering(.order2,#order2,u,v,switched),
                  undeclare(u,v,switched),
                  <adatr u, ... : integer>)]

apply -- [sd pre: (true)
```

```
                    comod: (all)
                      mod: (order2\pc)
                     post: (terminated(order2))]

      close -- 33 steps/applications

    join -- [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (terminated(order2),#stdout[1] le #stdout[2],
                      (#stdout[3] = 0 & #stdout[1] = stdin\237) &
                         #stdout[2] = stdin\239 or
                        (#stdout[3] = 1 & #stdout[1] = stdin\239) &
                          #stdout[2] = stdin\237)]

 close -- 15 steps/applications

<sdvs.3>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  (adatr "tutorial/order2.ada",
   prove order2.sd
     proof:
       (apply u(1),
        apply u(1),
        apply 6,
        go,
        cases .u le .v
          then proof:
            (apply 8,
             go ~(#u le #v))
          else proof: go terminated(order2)))
```

The first close is the "close" of the second case, and the second close is the "close" of the

proof of *order2.sd*.

7.1.4 Ada loops

The Ada programs in the first three Ada examples did not include a loop. A proof of correctness of an Ada program with a loop will almost always include the **induct** command, which we illustrate in our last example.

Example 17 Consider the Ada program add in the file add.ada.

```
with text_io; use text_io;
with integer_io; use integer_io;
procedure add is
   i,s,x,y : integer;
   begin
    get(x);
    get(y);
      i:= 0;
      s:= x;
      while i < y loop
        i:= i+1;
        s:= s+1;
      end loop;
    put(s);
  end add;
```

If the input for the object y is nonnegative, then the output of s is the sum of x and y. This is the assertion of the state delta *add.sd*:

```
  [sd pre: (ada(add.ada),.stdin[2] ge 0)
    comod: (all)
      mod: (all)
     post: (terminated(add),#stdout[1] = .stdin[1] + .stdin[2])]
```

We open the proof of *add.sd* and execute to the "while" loop using the **go** command with no parameters.

```
<sdvs.1>  adatr
   path name[tutorial/order2.ada]:  tutorial/add.ada

 Parsing Stage 3 Ada file -- "tutorial/add.ada"

 Translating Stage 3 Ada file -- "tutorial/add.ada"
```

```
<sdvs.2>  setflag
   flag variable:  autoclose
   on or off[off]:  on

 setflag autoclose -- on

<sdvs.3>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  prove
   state delta[]:  add.sd
   proof[]:  <CR>

 open -- [sd pre: (ada(add.ada),.stdin[2] ge 0)
          comod: (all)
            mod: (all)
           post: (terminated(add),
                  #stdout[1] = .stdin[1] + .stdin[2])]

 Complete the proof.

<sdvs.1.1>  go
   until[]:  <CR>

   apply -- [sd pre: (true)
             comod: (all)
               mod: (add\pc)
              post: (<adatr procedure add is
                                i, ... : integer
                              begin
                                get (x);
                                ...
                              end add;>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (add\pc,add)
              post: (alldisjoint(add,.add,i,s,x,y),
                     covering(#add,.add,i,s,x,y),
                     declare(i,type(integer)),declare(s,type(integer)),
```

```
                declare(x,type(integer)),declare(y,type(integer)),
                <adatr i, ... : integer>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,add)
           post: (alldisjoint(add,.add,get\item),
                  covering(#add,.add,get\item),
                  declare(get\item,type(polymorphic)),
                  <adatr get (x)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc)
           post: (#add\pc = at(standard.text_io.get),
                  <adatr get (x)>)]

apply -- [sd pre: (.add\pc = at(standard.text_io.get))
          comod: (all)
            mod: (add\pc,stdin\ctr,get\item)
           post: (#get\item = .stdin[.stdin\ctr],
                  #stdin\ctr = .stdin\ctr + 1,
                  #add\pc = exited(standard.text_io.get),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc)
           post: (#add\pc = exited(standard.text_io.get),
                  <adatr get (x)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,x)
           post: (#x = .get\item,
                  <adatr get (x)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,add,get\item)
           post: (covering(.add,#add,get\item),undeclare(get\item),
                  <adatr get (x)>)]

apply -- [sd pre: (true)
          comod: (all)
```

```
            mod: (add\pc,add)
           post: (alldisjoint(add,.add,get\item!2),
                  covering(#add,.add,get\item!2),
                  declare(get\item!2,type(polymorphic)),
                  <adatr get (y)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc)
           post: (#add\pc = at(standard.text_io.get),
                  <adatr get (y)>)]

apply -- [sd pre: (.add\pc = at(standard.text_io.get))
          comod: (all)
            mod: (add\pc,stdin\ctr,get\item!2)
           post: (#get\item!2 = .stdin[.stdin\ctr],
                  #stdin\ctr = .stdin\ctr + 1,
                  #add\pc = exited(standard.text_io.get),
                  <adatr null;>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc)
           post: (#add\pc = exited(standard.text_io.get),
                  <adatr get (y)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,y)
           post: (#y = .get\item!2,
                  <adatr get (y)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,add,get\item!2)
           post: (covering(.add,#add,get\item!2),
                  undeclare(get\item!2),
                  <adatr get (y)>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (add\pc,i)
           post: (#i = 0,
                  <adatr i := 0;>)]
```

```
   apply -- [sd pre: (true)
             comod: (all)
               mod: (add\pc,s)
              post: (#s = .x,
                     <adatr s := x;>)]

   go -- no more declarations or statements
```

<sdvs.1.17> *simp*

expression: *.x=.stdin[1] and .y=.stdin[2] and .i=0 and .s=.x and .y ge 0*

```
 true
```

<sdvs.1.17> *usable*

```
 u(1) [sd pre: (~(.i lt .y))
        comod: (all)
          mod: (add\pc)
         post: (<adatr while i < y

                                 i := i + 1;
                                 ...
                               end loop;>)]

 u(2) [sd pre: (.i lt .y)
        comod: (all)
          mod: (add\pc)
         post: (<adatr while i < y

                                 i := i + 1;
                                 ...
                               end loop;>)]


 No usable quantified formulas.
```

<sdvs.1.17> *applicable*

The translation of the "while" loop is the conjunction of the two usable state deltas, $u(1)$ and $u(2)$. But neither one is applicable, because neither precondition is necessarily true. To proceed, we must use the **cases** command. The simpler of the two cases is that $(i \not< y)$, because, in this case, the value of y is 0 and $s = x + y$. Thus, we enter this predicate to the **cases** command and **go** until `terminated(add)` is true.

<sdvs.1.17> *cases*

```
   case predicate:  ~(.i lt .y)

   cases -- ~(.i lt .y)

     open -- [sd pre: (~(.i lt .y))
              comod: (all)
                mod: (all)
               post: (terminated(add),
                      #stdout[1] = stdin\365 + stdin\363)]

<sdvs.1.17.1.1>  applicable

 u(1) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                              i := i + 1;
                              ...
                            end loop;>)]

<sdvs.1.17.1.1>  go
   until[]:  terminated(add)

       apply -- [sd pre: (~(.i lt .y))
                 comod: (all)
                   mod: (add\pc)
                  post: (<adatr while i < y

                                       i := i + 1;
                                       ...
                                     end loop;>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc,add)
                  post: (alldisjoint(add,.add,put\item),
                         covering(#add,.add,put\item),
                         declare(put\item,type(polymorphic)),
                         <adatr put (s)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc,put\item)
                  post: (#put\item = .s,
```

```
                    <adatr put (s)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc)
             post: (#add\pc = at(standard.text_io.put),
                    <adatr put (s)>)]

  apply -- [sd pre: (.add\pc = at(standard.text_io.put))
            comod: (all)
              mod: (add\pc,stdout[.stdout\ctr],stdout\ctr)
             post: (#stdout[.stdout\ctr] = .put\item,
                    #stdout\ctr = .stdout\ctr + 1,
                    #add\pc = exited(standard.text_io.put),
                    <adatr null;>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc)
             post: (#add\pc = exited(standard.text_io.put),
                    <adatr put (s)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc,add,put\item)
             post: (covering(.add,#add,put\item),
                    undeclare(put\item),
                    <adatr put (s)>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc,add,i,s,x,y)
             post: (covering(.add,#add,i,s,x,y),undeclare(i,s,x,y),
                    <adatr i, ... : integer>)]

  apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc)
             post: (terminated(add))]

close -- 9 steps/applications

open -- [sd pre: (~((~(.i lt .y))))
          comod: (all)
            mod: (all)
```

```
                post: (terminated(add),
                       #stdout[1] = stdin\365 + stdin\363)]
```

Complete the proof.

<sdvs.1.17.2.1> *usable*

```
u(1) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (all)
        post: (terminated(add),#stdout[1] = stdin\365 + stdin\363)]

u(2) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                              i := i + 1;
                              ...
                             end loop;>)]

u(3) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                              i := i + 1;
                              ...
                             end loop;>)]
```

No usable quantified formulas.

<sdvs.1.17.2.1> *applicable*

```
u(3) [sd pre: (.i lt .y)
      comod: (all)
        mod: (add\pc)
       post: (<adatr while i < y

                              i := i + 1;
                              ...
                             end loop;>)]
```

<sdvs.1.17.2.1> *letsd*

```
   name:  loopsd1
   state delta[]:  u
         number:  2

       letsd -- loopsd1 = u(2)

<sdvs.1.17.2.2>  letsd
   name:  loopsd2
   state delta[]:  u
         number:  3

       letsd -- loopsd2 = u(3)

<sdvs.1.17.2.3>  pp
   object:  sd
   state delta name:  loopsd1

 [sd pre: (~(.i lt .y))
   comod: (all)
     mod: (add\pc)
    post: (<adatr while i < y

                        i := i + 1;
                        ...
                      end loop;>)]
```

The proof of the first case has closed and the proof of the second has opened. The first usable state delta asserts that the first case leads to the goal. The conjunction of the second and third usable state deltas is the translation of the "while" loop. Notice that we have conveniently used the **letsd** command to label the two components of the loop. Only the third state delta, *loopsd2*, is applicable.

As in Example 8 we have to induct on the counter i, from $.i = 0$ to $.i = .y$ with a comodification list of x and y. But in this case, the induction invariant is trickier. It can not simply be $.s = .x + .i$, because at the step case proof there will be no state deltas to apply, since both *loopsd*1 and *loopsd*2 have `all` in their comodification lists. In fact, at the step case proof, *loopsd*2 must be applicable so that we may execute through the loop, i.e., increment i and s. So we must add it to the induction invariant. However, even this addition will not suffice. The addition of *loopsd*2 to the invariant will allow us to complete the induction. But after the "close" of the induction, $.s = .x + .y$ and *loopsd*2 will both be true, but *loopsd*2 will not be applicable, because its precondition will be false. If at the end of the induction proof, we also had *loopsd*1 as a usable state delta, then we would be able to proceed with the proof, because it would be applicable. Thus the invariant of the induction proof must also include *loopsd*1. Finally, the modification list parameter of the induction command must have *add\pc* as well as i and s.

```
<sdvs.1.17.2.3>  induct
    induction expression:  .i
                     from:  0
                       to:  .y
          invariant list[]:  .s=.x+.i,formula(loopsd1),formula(loopsd2)
     comodification list[]:  x,y
       modification list[]:  i,s,add\pc
             base proof[]:  <CR>
             step proof[]:  <CR>

       induction -- .i from 0 to .y

         open -- [sd pre: (true)
                  comod: (all)
                   post: (.s = .x + .i,
                          [sd pre: (~(.i lt .y))
                            comod: (all)
                              mod: (add\pc)
                             post: (<adatr while i < y

                                                  i := i + 1;
                                                  ...
                                                end loop;>)],
                         [sd pre: (.i lt .y)
                            comod: (all)
                              mod: (add\pc)
                             post: (<adatr while i < y

                                                  i := i + 1;
                                                  ...
                                                end loop;>)],
                         .i = 0)]

         close -- 0 steps/applications

         open -- [sd pre: (.i ge 0,.i lt .y,.s = .x + .i,
                          [sd pre: (~(.i lt .y))
                            comod: (all)
                              mod: (add\pc)
                             post: (<adatr while i < y

                                                  i := i + 1;
                                                  ...
                                                end loop;>)],
                          [sd pre: (.i lt .y)
```

```
                    comod: (all)
                      mod: (add\pc)
                     post: (<adatr while i < y

                                          i := i + 1;
                                          ...
                                        end loop;>)])
          comod: (x,y)
            mod: (i,s,add\pc)
           post: (#s = #x + #i,
                  [sd pre: (~(.i lt .y))
                    comod: (all)
                      mod: (add\pc)
                     post: (<adatr while i < y

                                          i := i + 1;
                                          ...
                                        end loop;>)],
                 [sd pre: (.i lt .y)
                    comod: (all)
                      mod: (add\pc)
                     post: (<adatr while i < y

                                          i := i + 1;
                                          ...
                                        end loop;>)],
                 #i = .i + 1)]
```

```
Complete the proof.
```

The base case of the induction proof has closed and the step case has opened. We proceed with **apply** interspersed with queries.

```
<sdvs.1.17.2.3.2.1>  usable

 u(1) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                           i := i + 1;
                           ...
                         end loop;>)]

 u(2) [sd pre: (~(.i lt .y))
```

```
        comod: (all)
          mod: (add\pc)
         post: (<adatr while i < y

                                i := i + 1;
                                ...
                              end loop;>)]
```

```
No usable quantified formulas.

<sdvs.1.17.2.3.2.1>  applicable

 u(1) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                               i := i + 1;
                               ...
                             end loop;>)]

<sdvs.1.17.2.3.2.1>  apply
   sd/number[highest applicable/once]:  <CR>

           apply -- [sd pre: (.i lt .y)
                     comod: (all)
                       mod: (add\pc)
                      post: (<adatr while i < y

                                              i := i + 1;
                                              ...
                                            end loop;>)]

<sdvs.1.17.2.3.2.2>  usable

 u(1) [sd pre: (true)
        comod: (all)
          mod: (add\pc,i)
         post: (#i = .i + 1,
                <adatr i := i + 1;>)]


No usable quantified formulas.
```

```
<sdvs.1.17.2.3.2.2>  apply
   sd/number[highest applicable/once]:  <CR>

          apply -- [sd pre: (true)
                    comod: (all)
                      mod: (add\pc,i)
                     post: (#i = .i + 1,
                            <adatr i := i + 1;>)]

<sdvs.1.17.2.3.2.3>  apply
   sd/number[highest applicable/once]:  <CR>

          apply -- [sd pre: (true)
                    comod: (all)
                      mod: (add\pc,s)
                     post: (#s = .s + 1,
                            <adatr s := s + 1;>)]

        close -- 3 steps/applications

       join induction cases -- [sd pre: (0 le .y)
                                comod: (all,x,y)
                                  mod: (i,s,add\pc)
                                 post: (#i = .y,#s = #x + #y,
                                        [sd pre: (~(.i lt .y))
                                          comod: (all)
                                            mod: (add\pc)
                                           post: (<adatr while i < y

                                                       i :=  ...;
                                                       ...
                                                   end loop;>)],
                                      [sd pre: (.i lt .y)
                                        comod: (all)
                                          mod: (add\pc)
                                         post: (<adatr while i < y

                                                       i :=  ...;
                                                       ...
                                                   end loop;>)])]

 Complete the proof.

<sdvs.1.17.2.4>  usable
```

```
u(1) [sd pre: (.i lt .y)
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                              i := i + 1;
                              ...
                            end loop;>)]

u(2) [sd pre: (~(.i lt .y))
       comod: (all)
         mod: (add\pc)
        post: (<adatr while i < y

                              i := i + 1;
                              ...
                            end loop;>)]


No usable quantified formulas.

<sdvs.1.17.2.4>  applicable

u(2) [sd pre: (~(.i lt .y))
      comod: (all)
        mod: (add\pc)
       post: (<adatr while i < y

                             i := i + 1;
                             ...
                           end loop;>)]

<sdvs.1.17.2.4>  simp
   expression:  .s=.x+.y and .x=.stdin[1] and .y=.stdin[2]

true
```

The first "close" was the end of the step case of the induction proof. The rest of the proof is now routine. We execute to the end using **go**.

```
<sdvs.1.17.2.4>  go
   until[]:  terminated(add)

      apply -- [sd pre: (~(.i lt .y))
                comod: (all)
```

```
              mod: (add\pc)
             post: (<adatr while i < y

                                    i := i + 1;
                                    ...
                                  end loop;>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc,add)
             post: (alldisjoint(add,.add,put\item),
                    covering(#add,.add,put\item),
                    declare(put\item,type(polymorphic)),
                    <adatr put (s)>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc,put\item)
             post: (#put\item = .s,
                    <adatr put (s)>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc)
             post: (#add\pc = at(standard.text_io.put),
                    <adatr put (s)>)]

apply -- [sd pre: (.add\pc = at(standard.text_io.put))
            comod: (all)
              mod: (add\pc,stdout[.stdout\ctr],stdout\ctr)
             post: (#stdout[.stdout\ctr] = .put\item,
                    #stdout\ctr = .stdout\ctr + 1,
                    #add\pc = exited(standard.text_io.put),
                    <adatr null;>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc)
             post: (#add\pc = exited(standard.text_io.put),
                    <adatr put (s)>)]

apply -- [sd pre: (true)
            comod: (all)
              mod: (add\pc,add,put\item)
             post: (covering(.add,#add,put\item),
```

```
                        undeclare(put\item),
                        <adatr put (s)>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc,add,i,s,x,y)
                  post: (covering(.add,#add,i,s,x,y),undeclare(i,s,x,y),
                         <adatr i, ... : integer>)]

       apply -- [sd pre: (true)
                 comod: (all)
                   mod: (add\pc)
                  post: (terminated(add))]

     close -- 12 steps/applications

   join -- [sd pre: (true)
            comod: (all)
              mod: (all)
             post: (terminated(add),
                    #stdout[1] = stdin\365 + stdin\363)]

 close -- 17 steps/applications

<sdvs.2>  quit

 Q.E.D.  The proof for this session is in 'sdvsproof'.

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  pp
   object:  proof
   proof name:  sdvsproof

 proof sdvsproof:

  prove add.sd
    proof:
      (go,
       cases ~(.i lt .y)
         then proof: go terminated(add)
         else proof:
           (letsd loopsd1 = u(2),
```

```
letsd loopsd2 = u(3),
induct on:      .i
  from:         0
  to:           .y
  invariants:   (.s = .x + .i,formula(loopsd1),
                 formula(loopsd2))
  comodlist:    (x,y)
  modlist:      (i,s,add\pc)
  base proof:
  step proof:
    (apply u(1),
     apply u(1),
     apply u(1)),
go terminated(add)))
```

7.2 VHDL

The following Stage 2 VHDL description, contained in the file `full_adder_dataflow.vhdl`, models a hardware device we call a *one-bit full adder*. It accepts three input bits, `x`, `y`, and `cin` ("carry-in"), which are to be added and the result recorded on output ports `sum` and `cout` ("carry-out"), also of type `BIT`. The modeling is in the *dataflow* style of register-transfer-like concurrent signal assignment statements.

The architecture uses an auxiliary signal `a`, which stores the logical exclusive or, `XOR`, of the `x` and `y` input ports; `a` is subsequently `XOR`-ed with input port `cin` to yield the value of the output port `sum`. The `cout` output port is set to bit '1' if any two of the three input ports are '1'.

The architecture body consists of three concurrent signal assignment statements, in which the explicit delays are arbitrarily chosen for illustrative purposes. The Stage 2 VHDL translator will regard each of these concurrent signal assignment statements as an equivalent `PROCESS` statement.

The header line in the VHDL code that follows represents our device in Stage 2 VHDL for giving a name to the whole hardware description; the tag `DESIGN_FILE` is not part of official VHDL syntax. This device will be eliminated once VHDL design libraries are interfaced with SDVS.

```
DESIGN_FILE adder IS


ENTITY full_adder IS

  PORT ( x, y, cin : IN  BIT;
         sum, cout : OUT BIT );

END full_adder;


ARCHITECTURE dataflow OF full_adder IS

  SIGNAL a : BIT;

BEGIN

  update_a :
    a <= x XOR y AFTER 3 NS;

  update_sum :
    sum <= a XOR cin AFTER 5 NS;
```

```
    update_cout :
      cout <= (x AND y)   OR
              (x AND cin) OR
              (y AND cin) AFTER 7 NS;
END dataflow;
```

7.2.1 State Delta specification

We wish to formulate and prove the following claim about the VHDL description `adder`:

> At any point at which the translation of `full_adder_dataflow.vhdl` holds, there will be a point at which the model will have been elaborated and such that at some later point, the values of the `sum` and `cout` signals will reflect the sum of the input ports `x`, `y`, and `cin`; furthermore, at this point the model will be done executing.

This English-language specification is expressed as the following state delta, contained in the file `full_adder_dataflow.spec`:

```
full_adder_dataflow.sd =

  [sd pre: (vhdl(full_adder_dataflow.vhdl))
    comod:
      mod: (all)
     post: (vhdl_model_elaboration_complete(adder),
            [sd pre: (true)
              comod: (all)
                mod: (all)
               post: (|#cout @ #sum| = |.x ++ .y ++ .cin|,
                      vhdl_model_execution_complete(adder))])]
```

A bit in SDVS is represented as a bitstring of length one. The theory of bitstrings implemented by the Simplifier includes the operators `@` and `++`, denoting bitstring concatenation and bitstring addition, respectively. Furthermore, the operator `| |` denotes the integer value of its bitstring argument under unsigned radix-two arithmetic. Its use in the above specification is crucial: whereas the concatenation of two bitstrings of length one produces a bitstring of length two, the bitstring sum of three bitstrings of length one is (by definition) a bitstring of length three; however, in our case the integer value of both sides of the equation should be the same.

The most important general observation to make about the above specification is the appearance of a *nested state delta* in the postcondition of the top-level SD[13] `full_adder_dataflow.sd`,

[13] Henceforth, "SD" is an abbreviation for "state delta."

with the intuitively desired final state as the postcondition of the *nested* SD. This device is common to most SDVS VHDL specifications, reflecting the fact that *it is the passage from the precondition time to the postcondition time of the top-level SD that allows the places mentioned in the final (nested) postcondition to be created*, by elaboration of the corresponding declarations in the VHDL description. Referring to these places in advance of their creation, e.g. in the postcondition of the top-level SD, can result in false specifications in cases where the corresponding declarations contain initialization expressions.

The practical consequence of this structure for the specification is that during the proof, once symbolic execution has reached a point where all the declarations have been elaborated, it is necessary to open a proof of the nested SD, and this is the only point at which it is appropriate to do so.

7.2.2 Interactive proof development

We present the trace of an interactive SDVS proof session showing the construction of a proof that the VHDL description satisfies its specification. This trace is punctuated with various remarks elucidating typical aspects of VHDL correctness proofs. The reader is referred to [15] for a formal semantic specification of the Stage 2 VHDL language translator.

Our general proof strategy is to simulate the VHDL description with symbolic values, aiming to reach a state in which the final postcondition of the state delta specification `full_adder_dataflow.sd` is true. At points in the proof where no usable state deltas are known to be applicable, static reasoning (by invocation of suitable lemmas) will establish that certain preconditions do indeed hold, so that symbolic execution can proceed.

The salient aspects of the general correctness proof of the one-bit full adder, distinguishing it from mere simulation of the description with concrete values, are as follows:

- The initial values of the input ports `x`, `y`, and `cin` are *symbolic*, rather than concrete bit values.

  The *VHDL Language Reference Manual* (LRM) [5] specifies implicit default values for objects that lack an explicit default expression in their declarations (see, e.g., Section 4.3.1.2 of [5]). We conjecture that the rationale for this (rather odd) convention stems from the *simulation semantics* for VHDL as defined by the LRM: without *concrete* values for objects, a description cannot be *simulated* (in the usual sense of the word). On the other hand, for the purposes of verification, it is not at all suitable to assume implicit default values for uninitialized objects: by definition, a correctness proof must be valid for *arbitrary* values of (nonconstant) objects. Therefore, the VHDL translator assigns *symbolic values* to uninitialized objects.

- Symbolic values for the input ports imply two important consequences for the correctness proof:

  - During each execution cycle, when the VHDL translator updates signals and then determines which processes should resume, a case analysis must be made on

whether actual *events* occurred on signals to which processes are sensitive, that is, whether the updates actually *changed* those signals' values. Indeed, according to the LRM [5], a process that resumes execution by virtue of its sensitivity to a signal does so only as a result of such an event; "stuttering" on the old signal value is not enough.

- When, as the result of an inertial signal assignment statement, the projected output waveform on a signal's driver is updated with new transactions, a case analysis must be made on whether the signal value currently scheduled for the greatest time strictly less than the time of the earliest new transaction is, or is not, equal to the value of that new transaction. The VHDL LRM preemption rules for updating the projected output waveform distinguish between these cases (see [5], Section 8.3.1).

We begin by initializing the system and turning the `autoclose` flag to `off`, in order to develop the proof without SDVS closing it automatically.

```
<sdvs.1>  init
   proof name[]:  <CR>

State Delta Verification System, Version 11

Restricted to authorized users only.

<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[on]:  off

 setflag autoclose -- off
```

Our first essential order of business is to translate the VHDL description contained in file `full_adder_dataflow.vhdl` into `vhdl(full_adder_dataflow.vhdl)`, its state delta representation, so that we can prove our claim about the description. This is done by invoking the VHDL translator with the command `vhdltr`, giving it the source VHDL file as its argument.

```
<sdvs.2>  vhdltr
   path name[testproofs/foo.vhdl]:  testproofs/vhdl2/full_adder_dataflow.vhdl

 Parsing Stage 2 VHDL file -- "testproofs/vhdl2/full_adder_dataflow.vhdl"

 Translating Stage 2 VHDL file
   -- "testproofs/vhdl2/full_adder_dataflow.vhdl"

<sdvs.3>  pp
```

```
   object: vhdl
   file name[full_adder_dataflow.vhdl]: full˙adder˙dataflow.vhdl

 alldisjoint(adder,.adder)
 covering(.adder,adder\pc,vhdltime,vhdltime_previous)
 declare(vhdltime,type(vhdltime))
 declare(vhdltime_previous,type(vhdltime))
 .vhdltime = vhdltime(0,0)
 .vhdltime_previous = vhdltime(0,0)
 [sd pre: (true)
   comod: (all)
     mod: (adder\pc)
    post: (<VHDLTR>)]
```

We have just exhibited the "initial segment" of the translation of the full adder description, consisting of the declaration and initialization of the places `vhdltime` and `vhdltime_previous`, as well as a state delta whose postcondition contains a representation of (a state delta for) the incremental continuation of the translation.

In general, each state delta generated by the VHDL translator will contain, as part of its postcondition, a *continuation label* enclosed in angle brackets; this continuation label simply stands for the next state delta to be incrementally generated by the translator — the *continuation*. The generic label <VHDLTR> appears most frequently, but occasional labels attempt to be more descriptive of the next increment of translation.

Sometimes, as in the initial segment of translation, the translator generates a state delta with precondition `(true)`, comodlist `(all)`, a `(\pc)` modlist, and only a continuation in the postcondition. Such a state delta corresponds to an *action*, to be unconditionally performed by the translator, resulting in no change in the state (contents of places) except for the program counter. When such a state delta is applied, it is not printed out in its entirety in the proof trace; rather, the tag `action` is printed, followed by the continuation label.

```
<sdvs.3> read
   path name[testproofs/foo.proofs]: testproofs/vhdl2/full˙adder˙dataflow.spec

 Definitions read from file "testproofs/vhdl2/full_adder_dataflow.spec"
   -- (full_adder_dataflow.sd,full_adder_dataflow_original.sd)

<sdvs.4> ppsd
   state delta: full˙adder˙dataflow.sd

 [sd pre: (vhdl(full_adder_dataflow.vhdl))
     mod: (all)
    post: (vhdl_model_elaboration_complete(adder),
           [sd pre: (true)
```

```
        comod: (all)
          mod: (all)
         post: (|#cout @ #sum| = |(.x ++ .y) ++ .cin|,
                vhdl_model_execution_complete(adder))])]
```

This is the specification to be proved.

The proof will require two lemmas concerning bitstrings, which we read from a file and display.

```
<sdvs.4>  read
   path name[testproofs/vhdl2/full_adder_dataflow.spec]:
      testproofs/vhdl2/full_adder_dataflow.lemmas

 Definitions read from file "testproofs/vhdl2/full_adder_dataflow.lemmas"
   -- (append_cout_sum.lemma)

<sdvs.5>  pp
   object:  append_cout_sum.lemma

 lemma append_cout_sum.lemma (x,y,cin,sum,cout):
   (((((lh(x) = 1 &
         lh(y) = 1) &
           lh(cin) = 1) &
             lh(sum) = 1) &
               lh(cout) = 1) &
                 sum = (x usxor y) usxor cin) &
                   cout = (x && y usor x && cin) usor y && cin
     --> |cout @ sum| = |(x ++ y) ++ cin|
```

Lemma `append_cout_sum.lemma` asserts that if bits `sum` and `cout` are related to bits `x`, `y`, and `cin` as indicated, then the bitstring concatenation of `cout` with `sum` has the same integer value as the bitstring sum of `x`, `y` and `cin`. Again, this lemma has an easy proof by exhaustive case analysis of the possibilities for `x`, `y`, and `cin`.

Note that this lemma essentially mimics the way in which the VHDL description computes `sum` and `cout`; not surprisingly, it will be used to establish the required static fact upon completion of the dynamic symbolic execution of the description.

We now open the proof of `full_adder_dataflow.sd`:

```
<sdvs.5>  prove
   state delta[]:  full_adder_dataflow.sd
   proof[]:  <CR>
```

```
open -- [sd pre: (vhdl(full_adder_dataflow.vhdl))
            mod: (all)
           post: (vhdl_model_elaboration_complete(adder),
                  [sd pre: (true)
                    comod: (all)
                      mod: (all)
                     post: (|#cout @ #sum| = |(.x ++ .y) ++ .cin|,
                            vhdl_model_execution_complete(adder))])]

Complete the proof.

<sdvs.5.1>  nsd

[sd pre: (true)
  comod: (all)
    mod: (adder\pc)
   post: (<VHDLTR>)]
```

The applicable state delta just shown is the "bootstrap" state delta for the incremental translation of the Stage 2 VHDL description. Issuing the command go with the until argument of vhdl_model_elaboration_complete(adder) will apply this state delta as the first in a sequence of continuations that accomplish automatic elaboration of the entity port declarations for x, y, cin, sum, and cout, as well as of the internal signal a in the architecture body and the processes represented by the three concurrent signal assignment statements.

```
<sdvs.5.1> go
   until[]: vhdl_model_elaboration_complete(adder)

   action -- <VHDLTR>

   apply -- [sd pre: (true)
              comod: (all)
                mod: (adder,adder)
               post: (alldisjoint(adder,.adder,x,y,cin,driver\x,driver\y,
                                  driver\cin),
                      covering(#adder,.adder,x,y,cin,driver\x,driver\y,
                               driver\cin),
                      declare(x,type(bit)),
                      declare(driver\x,type(waveform,type(bit))),
                      declare(x,type(fn,val(.driver\x,.vhdltime))),
                      declare(y,type(bit)),
                      declare(driver\y,type(waveform,type(bit))),
                      declare(y,type(fn,val(.driver\y,.vhdltime))),
                      declare(cin,type(bit)),
```

```
                declare(driver\cin,type(waveform,type(bit))),
                declare(cin,type(fn,val(.driver\cin,.vhdltime))),
                <VHDLTR>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder,x,y,cin,driver\x,driver\y,driver\cin)
           post: (#driver\x
                    = waveform(x,transaction(vhdltime(0,0),x\13)),
                  #driver\y
                    = waveform(y,transaction(vhdltime(0,0),y\15)),
                  #driver\cin
                    = waveform(cin,
                               transaction(vhdltime(0,0),cin\17)),
                  <VHDLTR>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder,adder)
           post: (alldisjoint(adder,.adder,sum,cout,driver\sum,
                              driver\cout),
                  covering(#adder,.adder,sum,cout,driver\sum,
                           driver\cout),
                  declare(sum,type(bit)),
                  declare(driver\sum,type(waveform,type(bit))),
                  declare(sum,type(fn,val(.driver\sum,.vhdltime))),
                  declare(cout,type(bit)),
                  declare(driver\cout,type(waveform,type(bit))),
                  declare(cout,type(fn,val(.driver\cout,.vhdltime))),
                  <VHDLTR>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder,sum,cout,driver\sum,driver\cout)
           post: (#driver\sum
                    = waveform(sum,
                               transaction(vhdltime(0,0),sum\22)),
                  #driver\cout
                    = waveform(cout,
                               transaction(vhdltime(0,0),cout\24)),
                  <VHDLTR>)]

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder,adder)
```

```
                post: (alldisjoint(adder,.adder,a,driver\a),
                       covering(#adder,.adder,a,driver\a),
                       declare(a,type(bit)),
                       declare(driver\a,type(waveform,type(bit))),
                       declare(a,type(fn,val(.driver\a,.vhdltime))),
                       <VHDLTR>)]

   apply -- [sd pre: (true)
             comod: (all)
               mod: (adder,a,driver\a)
              post: (#driver\a
                        = waveform(a,transaction(vhdltime(0,0),a\29)),
                       <VHDLTR>)]

   action -- <ELABORATE PROCESS: UPDATE_A>

   action -- <ELABORATE PROCESS: UPDATE_SUM>

   action -- <ELABORATE PROCESS: UPDATE_COUT>

   go -- breakpoint reached

<sdvs.5.11>
```

The evaluation of the three SDVS commands **vhdltime**, **vhdl-signals**, and **vhdl-processes** is a convenient means of querying SDVS about aspects of the state of the Stage 2 VHDL proof. Particularly in the case of signals, this query provides information in a much more intelligible form than that returned by, say, the query command **ppl** . Note that `0(1)` is the Simplifier representation of the bitstring with integer value 0 and length 1 — that is, the bit `'0'`.

```
<sdvs.5.11>  vhdltime


         global time  =  0

         delta  time  =  0


<sdvs.5.11>  vhdl-signals
   signal-names[all]:  <CR>
      simplify?[no]:  <CR>


         signal X :
```

```
  current value    =  x\13
  previous value   =  x\13
  projected output waveform =  ()
  driver history   =  (transaction(vhdltime(0,0),x\13))

signal Y :
  current value    =  y\15
  previous value   =  y\15
  projected output waveform =  ()
  driver history   =  (transaction(vhdltime(0,0),y\15))

signal CIN :
  current value    =  cin\17
  previous value   =  cin\17
  projected output waveform =  ()
  driver history   =  (transaction(vhdltime(0,0),cin\17))

signal SUM :
  current value    =  sum\22
  previous value   =  sum\22
  projected output waveform =  ()
  driver history   =  (transaction(vhdltime(0,0),sum\22))

signal COUT :
```

```
    current value    =  cout\24

    previous value   =  cout\24

    projected output waveform =  ()

    driver history   =  (transaction(vhdltime(0,0),cout\24))


  signal A :

    current value    =  a\29

    previous value   =  a\29

    projected output waveform =  ()

    driver history   =  (transaction(vhdltime(0,0),a\29))
```

The declarations have been elaborated symbolically. For example, places `x` and `driver\x` have been created to represent a signal and its driver, respectively, and the contents of `driver\x` have been initialized with `waveform(x,transaction(vhdltime(0,0),x\13))`, a waveform (indexed by `x`) consisting of a single transaction. This transaction stipulates that at `vhdltime(0,0)`, `x` acquires the symbolic bit value `x\13`.

In the display generated by the command `vhdl-signals`, the driver is split conceptually into two disjoint parts, each represented as a list:

- A *projected output waveform*, consisting of future transactions scheduled to occur on the signal (some of which might be *preempted*, or deleted from the waveform, during subsequent execution of the description). The time components of projected transactions are all greater than the `vhdltime`. For ease of reference, the projected transactions are displayed in chronological order according to their time components, so that the next scheduled transaction occurs first in the list.

- A *driver history*, consisting of those transactions that have already been "actualized," i.e., whose time component is at most the value `.vhdltime`. For ease of reference once again, but in contradistinction to the projected output waveform, these transactions are displayed in reverse chronological order: the most recent actualized transaction for the signal appears at the head of the driver history, and its value component is always the current value of the signal driver.

Thus, the entire signal driver itself is the concatenation of the reverse of the driver history with the projected output waveform.

```
<sdvs.5.11>  vhdl-processes
   process-names[all]:  <CR>

          process UPDATE_A :

            current state       =  SUSPENDED

            scheduled time      =  VHDLTIME(0,0)

            scheduled reason    =  INITIALIZATION


          process UPDATE_SUM :

            current state       =  SUSPENDED

            scheduled time      =  VHDLTIME(0,0)

            scheduled reason    =  INITIALIZATION


          process UPDATE_COUT :

            current state       =  SUSPENDED

            scheduled time      =  VHDLTIME(0,0)

            scheduled reason    =  INITIALIZATION
```

Note that the Stage 2 VHDL translator represents the three concurrent signal assignment statements as processes.

All processes are shown as currently suspended, because we have not yet begun executing the model, but they are scheduled to "resume" execution at `vhdltime(0,0)`, by reason of the *initialization phase* of the simulation semantics informally defined in the VHDL LRM [5]. In the initialization phase, each process is executed until it suspends. As the next applicable state delta indicates, the translation is ready to commence model execution.

```
<sdvs.5.11>  nsd

 [sd pre: (true)
   comod: (all)
     mod: (adder\pc)
    post: (<BEGIN VHDL MODEL EXECUTION>)]
```

```
<sdvs.5.11>  whynotgoal
   simplify?[no]:  <CR>

 g(2) [sd pre: (true)
       comod: (all)
         mod: (all)
        post: (|#cout @ #sum| = |(.x ++ .y) ++ .cin|,
               vhdl_model_execution_complete(adder))]
```

This is an appropriate point at which to open a proof of the goal g(2).

```
<sdvs.5.11>  prove
   state delta[]:  g
         number:  2
   proof[]:  <CR>

   open -- [sd pre: (true)
            comod: (all)
              mod: (all)
             post: (|#cout @ #sum| = |(.x ++ .y) ++ .cin|,
                    vhdl_model_execution_complete(adder))]

 Complete the proof.
```

Applying the next and subsequent applicable state deltas causes each process to execute, in order, and then suspend.

```
<sdvs.5.11.1>  apply
   sd/number[highest applicable/once]:  4

     action -- <BEGIN VHDL MODEL EXECUTION>

     action -- <BEGIN INITIALIZATION PHASE>

     action -- <... INITIALIZATION PHASE: EACH PROCESS EXECUTES UNTIL SUSPENSION>

     action -- <EXECUTE PROCESS: UPDATE_A>

<sdvs.5.11.5>  usable

 u(1) [sd pre: (~(preemption(.driver\a,
                             transaction(timeplus(.vhdltime,
                                                  vhdltime(3000000,0)),
```

```
                                        .x usxor .y))))
        comod: (all)
          mod: (adder\pc,driver\a)
         post: (#driver\a
                    = inertial_update(.driver\a,
                                      transaction(timeplus(.vhdltime,
                                                           vhdltime(3000000,0)),
                                                  .x usxor .y)),
                <VHDLTR>)]

u(2) [sd pre: (preemption(.driver\a,
                          transaction(timeplus(.vhdltime,
                                               vhdltime(3000000,0)),
                                      .x usxor .y)))
        comod: (all)
          mod: (adder\pc,driver\a)
         post: (#driver\a
                    = inertial_update(.driver\a,
                                      transaction(timeplus(.vhdltime,
                                                           vhdltime(3000000,0)),
                                                  .x usxor .y)),
                <VHDLTR>)]


No usable quantified formulas.
```

The above pair of usable SDs constitute the state delta semantics of the signal assignment `a <= x XOR y AFTER 3 NS` in the body of process `update_a`; it is important to understand the rationale for this. The VHDL semantics of *inertial* driver update (the default being used here, as opposed to *transport* update, which must be explicitly specified in the signal assignment statement) requires that the inertial update take into account whether or not currently scheduled transactions on the projected output waveform are to be *preempted* (replaced) by the new transaction(s) to be scheduled ([5], Section 8.3.1). Thus, the translation of the signal assignment statement generates a conjunction of two SDs, each predicated in its precondition on the occurrence of preemption or not. Note that the conjunction of the preconditions is simply `true`, meaning that the update will occur in any case; the only difference will be in the transactions of the projected output waveform following the update.

In the present situation, the projected output waveform of signal `a` is empty, so only state delta `u(1)` is applicable. However, we will have occasion to revisit this issue later on in a less obvious situation.

Observe also how the Stage 2 VHDL translator converts all VHDL `TIME` units to *femtoseconds*, so that 3 nanoseconds (`3 NS`) is represented as 3000000 femtoseconds.

Invoking the SDVS go command causes successive next-applicable state deltas to be applied

until the current goal is reached or the top usable state delta is not applicable (or until an explicitly stated condition is reached).

```
<sdvs.5.11.5>  go
   until[]:  <CR>

     apply -- [sd pre: (~(preemption(.driver\a,
                                     transaction(timeplus(.vhdltime,
                                                          vhdltime(3000000,0)),
                                                 .x usxor .y))))
                comod: (all)
                  mod: (adder\pc,driver\a)
                 post: (#driver\a
                          = inertial_update(.driver\a,
                                            transaction(timeplus(.vhdltime,
                                                                 vhdltime(3000000,0))
                                                        .x usxor .y)),
                        <VHDLTR>)]

     action -- <SUSPEND PROCESS: UPDATE_A>

     action -- <... INITIALIZATION PHASE: EACH PROCESS EXECUTES UNTIL SUSPENSION>

     action -- <EXECUTE PROCESS: UPDATE_SUM>

     apply -- [sd pre: (~(preemption(.driver\sum,
                                     transaction(timeplus(.vhdltime,
                                                          vhdltime(5000000,0)),
                                                 .a usxor .cin))))
                comod: (all)
                  mod: (adder\pc,driver\sum)
                 post: (#driver\sum
                          = inertial_update(.driver\sum,
                                            transaction(timeplus(.vhdltime,
                                                                 vhdltime(5000000,0)),
                                                        .a usxor .cin)),
                        <VHDLTR>)]

     action -- <SUSPEND PROCESS: UPDATE_SUM>

     action -- <... INITIALIZATION PHASE: EACH PROCESS EXECUTES UNTIL SUSPENSION>

     action -- <EXECUTE PROCESS: UPDATE_COUT>

     apply -- [sd pre: (~(preemption(.driver\cout,
```

```
                                    transaction(timeplus(.vhdltime,
                                                         vhdltime(7000000,0)),
                                                (.x && .y usor
                                                    .x && .cin) usor
                                                  .y && .cin))))
                comod: (all)
                  mod: (adder\pc,driver\cout)
                 post: (#driver\cout
                           = inertial_update(.driver\cout,
                                             transaction(timeplus(.vhdltime,
                                                                  vhdltime(7000000,0)),
                                                         (.x && .y usor
                                                             .x && .cin) usor
                                                           .y && .cin)),
                        <VHDLTR>)]

     action -- <SUSPEND PROCESS: UPDATE_COUT>

     action -- <END INITIALIZATION PHASE>
```

Having completed the initialization phase of execution, the VHDL translator determines the earliest future time at which a signal driver becomes active (i.e., has a transaction on its projected output waveform) or a process is scheduled to resume (by reason of timeout or sensitivity to a signal). This earliest time, if it exists, is the one to which vhdltime is advanced, initiating a new execution cycle: signals are updated and processes (possibly) resumed [5].

```
     action -- <BEGIN EXECUTION CYCLE:
                  1. ADVANCE EXECUTION TIME,
                  2. UPDATE SIGNALS,
                  3. RESUME PROCESSES>

     apply -- [sd pre: (true)
                comod: (all)
                  mod: (adder\pc,vhdltime,vhdltime_previous,a)
                 post: (#vhdltime = vhdltime(3000000,0),
                        #vhdltime_previous = .vhdltime,
                        <UPDATE SIGNALS>)]

     action -- <RESUME (?) NEXT SCHEDULED PROCESS: UPDATE_SUM>

     go -- no more declarations or statements
```

<sdvs.5.11.19> *vhdltime*

```
        global time  =  3000000

        delta  time  =  0
```

```
<sdvs.5.11.19>  vhdl-signals
   signal-names[all]:  a, sum, cout
       simplify?[no]:  yes
```

```
        signal A :

          current value     =  x\13 usxor y\15

          previous value    =  a\29

          projected output waveform =  ()

          driver history    =  (transaction(vhdltime(3000000,0),
                                            x\13 usxor y\15),
                                transaction(vhdltime(0,0),a\29))


        signal SUM :

          current value     =  sum\22

          previous value    =  sum\22

          projected output waveform =  (transaction(vhdltime(5000000,0),
                                                    a\29 usxor cin\17))

          driver history    =  (transaction(vhdltime(0,0),sum\22))


        signal COUT :

          current value     =  cout\24

          previous value    =  cout\24

          projected output waveform =  (transaction(vhdltime(7000000,0),
                                                    (x\13 && y\15 usor
                                                     x\13 && cin\17) usor
```

```
                                                    y\15 && cin\17))

          driver history   =  (transaction(vhdltime(0,0),cout\24))
```

<sdvs.5.11.19> *vhdl-processes*
process-names[all]: <*CR*>

```
        process UPDATE_A :

          current state      =  SUSPENDED


        process UPDATE_SUM :

          current state      =  SUSPENDED

          scheduled time     =  VHDLTIME(3000000,0)

          scheduled reason   =  SENSITIVITY


        process UPDATE_COUT :

          current state      =  SUSPENDED
```

Note that the query vhdl-processes reveals that of the three processes, *only* update_sum might resume execution at any later time. This is as it should be, in light of the following facts:

- the other two processes are sensitive only to the input signals; and
- we are operating under the implicit *stability assumption* that the input signals do not change for the *settle time* of the description [19].

<sdvs.5.11.19> *usable*

```
 u(1) [sd pre: (.a = val(.driver\a,.vhdltime_previous),
                .cin = val(.driver\cin,.vhdltime_previous))
       comod: (all)
         mod: (adder\pc)
        post: (<END EXECUTION CYCLE>)]

 u(2) [sd pre: (.cin ~= val(.driver\cin,.vhdltime_previous))
```

```
        comod: (all)
          mod: (adder\pc)
         post: ([sd pre: (true)
                 comod: (all)
                   mod: (adder\pc)
                  post: (<EXECUTE PROCESS: UPDATE_SUM>)])]

 u(3) [sd pre: (.a ~= val(.driver\a,.vhdltime_previous))
        comod: (all)
          mod: (adder\pc)
         post: ([sd pre: (true)
                 comod: (all)
                   mod: (adder\pc)
                  post: (<EXECUTE PROCESS: UPDATE_SUM>)])]


 No usable quantified formulas.

<sdvs.5.11.19>  nsd

 No applicable state deltas.

<sdvs.5.11.19>  whynotapply
   state delta[ highest usable]:  u
                         number:  1

 Because the following is not known to be true --

   .a = val(.driver\a,.vhdltime_previous)
```

This is the first crucial point to understand in the symbolic execution, as it relies on an important aspect of VHDL semantics. The essential point to realize is that the resumption of the process update_sum is contingent upon whether or not the signal a, to which the process is sensitive, has actually received a *new and different* value at the current time, vhdltime(3000000,0) (the value of signal cin will necessarily remain unchanged, as this signal is a port of mode IN). The VHDL semantics of process resumption requires that such an *event* on a must have occurred in order for update_sum to resume execution.

Thus, in order to render one of the two usable state deltas applicable, we must open up an argument by cases at this point.

```
<sdvs.5.11.19>  cases
   case predicate:  .a = val(.driver\a,.vhdltime'previous)

     cases -- .a = val(.driver\a,.vhdltime_previous)
```

```
        open -- [sd pre: (.a = val(.driver\a,.vhdltime_previous))
                  comod: (all)
                    mod: (all)
                   post: (|#cout @ #sum|
                             = |(x\36 ++ y\37) ++ cin\38|,
                           vhdl_model_execution_complete(adder))]
```

<sdvs.5.11.19.1.1> *apply*
sd/number[highest applicable/once]: *3*

```
        apply -- [sd pre: (.a = val(.driver\a,.vhdltime_previous),
                           .cin = val(.driver\cin,.vhdltime_previous))
                  comod: (all)
                    mod: (adder\pc)
                   post: (<END EXECUTION CYCLE>)]

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc,vhdltime,vhdltime_previous,sum)
                   post: (#vhdltime = vhdltime(5000000,0),
                          #vhdltime_previous = .vhdltime,
                          <UPDATE SIGNALS>)]
```

Note that, in this case, process update_sum did not resume; instead, a new execution cycle commenced and vhdltime advanced to the next time at which a signal had a transaction on its projected output waveform — this signal is sum, and the time is vhdltime(5000000,0).

<sdvs.5.11.19.1.4> *vhdltime*

```
        global time  =  5000000

        delta  time  =  0
```

<sdvs.5.11.19.1.4> *vhdl-signals*
signal-names[all]: *a, sum, cout*
simplify?[no]: *yes*

```
signal A :

  current value     =  a\29

  previous value    =  a\29

  projected output waveform =  ()

  driver history    =  (transaction(vhdltime(3000000,0),a\29),
                        transaction(vhdltime(0,0),a\29))


signal SUM :

  current value     =  a\29 usxor cin\17

  previous value    =  sum\22

  projected output waveform =  ()

  driver history    =  (transaction(vhdltime(5000000,0),
                                    a\29 usxor cin\17),
                        transaction(vhdltime(0,0),
                                    sum\22))


signal COUT :

  current value     =  cout\24

  previous value    =  cout\24

  projected output waveform =  (transaction(vhdltime(7000000,0),
                                            (x\13 && y\15 usor
                                             x\13 && cin\17) usor
                                             y\15 && cin\17))

  driver history    =  (transaction(vhdltime(0,0),cout\24))
```

<sdvs.5.11.19.1.4> *vhdl-processes*
process-names[all]: <*CR*>

```
        process UPDATE_A :

          current state      =  SUSPENDED


        process UPDATE_SUM :

          current state      =  SUSPENDED


        process UPDATE_COUT :

          current state      =  SUSPENDED


<sdvs.5.11.19.1.4>  apply
   sd/number[highest applicable/once]:  3

        action -- <END EXECUTION CYCLE>

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc,vhdltime,vhdltime_previous,cout)
                   post: (#vhdltime = vhdltime(7000000,0),
                          #vhdltime_previous = .vhdltime,
                          <UPDATE SIGNALS>)]

<sdvs.5.11.19.1.7>  vhdltime


        global time  =  7000000

        delta  time  =  0


<sdvs.5.11.19.1.7>  vhdl-signals
   signal-names[all]:  a, sum, cout
      simplify?[no]:  yes
```

```
signal A :

  current value    =  a\29

  previous value   =  a\29

  projected output waveform =  ()

  driver history   =  (transaction(vhdltime(3000000,0),a\29),
                       transaction(vhdltime(0,0),a\29))


signal SUM :

  current value    =  a\29 usxor cin\17

  previous value   =  sum\22

  projected output waveform =  ()

  driver history   =  (transaction(vhdltime(5000000,0),
                                    a\29 usxor cin\17),
                       transaction(vhdltime(0,0),
                                    sum\22))


signal COUT :

  current value    =  (x\13 && y\15 usor
                       x\13 && cin\17) usor
                       y\15 && cin\17

  previous value   =  cout\24

  projected output waveform =  ()

  driver history   =  (transaction(vhdltime(7000000,0),
                                    (x\13 && y\15 usor
                                     x\13 && cin\17) usor
                                     y\15 && cin\17),
                       transaction(vhdltime(0,0),
                                    cout\24))
```

<sdvs.5.11.19.1.7> *vhdl-processes*

```
process-names[all]: <CR>

        process UPDATE_A :

          current state      =  SUSPENDED


        process UPDATE_SUM :

          current state      =  SUSPENDED


        process UPDATE_COUT :

          current state      =  SUSPENDED
```

At this point, no signal drivers are active, and no processes are scheduled to resume. Therefore, the VHDL model will suspend execution indefinitely (that is, until it receives a new value for an input port).

```
<sdvs.5.11.19.1.7>  apply
   sd/number[highest applicable/once]:  4

        action -- <END EXECUTION CYCLE>

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>

        action -- <END VHDL MODEL EXECUTION>

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc)
                   post: (vhdl_model_execution_complete(adder))]

<sdvs.5.11.19.1.11>  whynotgoal
   simplify?[no]:  <CR>

 g(1) |#cout @ #sum| = |(x\36 ++ y\37) ++ cin\38|
```

The goal g(1) is established by a static proof that appeals directly to the lemma append_cout_sum.lemma:

```
<sdvs.5.11.19.1.11>  provebylemma
   formula to prove:  |.cout @ .sum| = |(.x ++ .y) ++ .cin|
       lemma name[]:  <CR>

         provebylemma append_cout_sum.lemma -- |.cout @ .sum|
                                                 = |(.x ++ .y) ++ .cin|

<sdvs.5.11.19.1.12>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.5.11.19.1.12>  close

       close -- 11 steps/applications

       open -- [sd pre: (~(.a = val(.driver\a,.vhdltime_previous)))
                comod: (all)
                  mod: (all)
                 post: (|#cout @ #sum|
                           = |(x\36 ++ y\37) ++ cin\38|,
                        vhdl_model_execution_complete(adder))]

 Complete the proof.
```

The second case has now been opened, wherein the current and previous values of signal a are asserted to be different.

Note that the `vhdltime` has reverted back to `vhdltime(3000000,0)`, and the signals have reverted to their states at the beginning of the previous case, except that the current value of signal a this time is the bit '1', represented in the Simplifier as the bitstring `1(1)` with integer value 1 and length 1.

```
<sdvs.5.11.19.2.1>  vhdltime

         global time  =  3000000

         delta  time  =  0

<sdvs.5.11.19.2.1>  vhdl-signals
   signal-names[all]:  a, sum, cout
       simplify?[no]:  yes
```

```
signal A :

  current value     =  a\86

  previous value    =  a\29

  projected output waveform =  ()

  driver history    =  (transaction(vhdltime(3000000,0),a\86),
                        transaction(vhdltime(0,0),a\29))


signal SUM :

  current value     =  sum\22

  previous value    =  sum\22

  projected output waveform =  (transaction(vhdltime(5000000,0),
                                            a\29 usxor cin\17))

  driver history    =  (transaction(vhdltime(0,0),sum\22))


signal COUT :

  current value     =  cout\24

  previous value    =  cout\24

  projected output waveform =  (transaction(vhdltime(7000000,0),
                                            (x\13 && y\15 usor
                                             x\13 && cin\17) usor
                                             y\15 && cin\17))

  driver history    =  (transaction(vhdltime(0,0),cout\24))
```

<sdvs.5.11.19.2.1> *vhdl-processes*
process-names[all]: <*CR*>

```
process UPDATE_A :
```

```
    current state      =  SUSPENDED


  process UPDATE_SUM :

    current state      =  SUSPENDED

    scheduled time     =  VHDLTIME(3000000,0)

    scheduled reason   =  SENSITIVITY


  process UPDATE_COUT :

    current state      =  SUSPENDED
```

Since the current cases branch presumes an event on the signal a, the scheduled process update_sum does resume execution.

```
<sdvs.5.11.19.2.1>  nsd

 [sd pre: (.a ~= val(.driver\a,.vhdltime_previous))
   comod: (all)
     mod: (adder\pc)
    post: ([sd pre: (true)
              comod: (all)
                mod: (adder\pc)
               post: (<EXECUTE PROCESS: UPDATE_SUM>)])]

<sdvs.5.11.19.2.1>  go
   until[]:  <CR>

        apply -- [sd pre: (.a ~= val(.driver\a,.vhdltime_previous))
                  comod: (all)
                    mod: (adder\pc)
                   post: ([sd pre: (true)
                             comod: (all)
                               mod: (adder\pc)
                              post: (<EXECUTE PROCESS: UPDATE_SUM>)])]

        action -- <EXECUTE PROCESS: UPDATE_SUM>

     go -- no more declarations or statements

<sdvs.5.11.19.2.3>  usable
```

```
u(1) [sd pre: (~(preemption(.driver\sum,
                            transaction(timeplus(.vhdltime,
                                                 vhdltime(5000000,0)),
                                        .a usxor .cin))))
       comod: (all)
         mod: (adder\pc,driver\sum)
        post: (#driver\sum
                  = inertial_update(.driver\sum,
                                    transaction(timeplus(.vhdltime,
                                                         vhdltime(5000000,0)),
                                                .a usxor .cin)),
              <VHDLTR>)]

u(2) [sd pre: (preemption(.driver\sum,
                          transaction(timeplus(.vhdltime,
                                               vhdltime(5000000,0)),
                                      .a usxor .cin)))
       comod: (all)
         mod: (adder\pc,driver\sum)
        post: (#driver\sum
                  = inertial_update(.driver\sum,
                                    transaction(timeplus(.vhdltime,
                                                         vhdltime(5000000,0)),
                                                .a usxor .cin)),
              <VHDLTR>)]


No usable quantified formulas.

<sdvs.5.11.19.2.3>  whynotapply
   state delta[ highest usable]:  u
                         number:  2

Because the following is not known to be true --

   preemption(.driver\sum,
              transaction(timeplus(.vhdltime,vhdltime(5000000,0)),
                          .a usxor .cin))
```

We have arrived at the second crucial point to understand in the symbolic execution; it revisits the earlier discussion of the preemptive semantics of inertial driver update.

The single action of process update_sum is to update the driver of signal sum, and the manner in which this update takes place depends on whether or not the value of the existing transaction on that driver's projected output waveform, namely cin\17 (= .cin), is or is not

equal to the value to be scheduled by the update transaction, namely `.a usxor .cin`. The semantics of inertial driver update in VHDL requires that the former (existing) transaction be deleted if these values are different (*preemption*), but retained if they are the same.

Thus, we again need to open a proof by cases at this juncture on whether or not preemption will take place:

```
<sdvs.5.11.19.2.3> cases
   case predicate: preemption(.driver\sum,
                              transaction(timeplus(.vhdltime,
                                                   vhdltime(5000000,0)),
                                          .a usxor .cin))

         cases -- preemption(.driver\sum,
                             transaction(timeplus(.vhdltime,
                                                  vhdltime(5000000,0)),
                                         .a usxor .cin))

           open -- [sd pre: (preemption(.driver\sum,
                                        transaction(timeplus(.vhdltime,
                                                             vhdltime(5000000,0)),
                                                    .a usxor .cin)))
                    comod: (all)
                      mod: (all)
                     post: (|#cout @ #sum|
                               = |(x\36 ++ y\37) ++ cin\38|,
                              vhdl_model_execution_complete(adder))]

<sdvs.5.11.19.2.3.1.1>  nsd

 [sd pre: (preemption(.driver\sum,
                      transaction(timeplus(.vhdltime,
                                           vhdltime(5000000,0)),
                                  .a usxor .cin)))
   comod: (all)
     mod: (adder\pc,driver\sum)
    post: (#driver\sum
             = inertial_update(.driver\sum,
                               transaction(timeplus(.vhdltime,
                                                    vhdltime(5000000,0)),
                                           .a usxor .cin)),
           <VHDLTR>)]

<sdvs.5.11.19.2.3.1.1>  apply
   sd/number[highest applicable/once]:  <CR>
```

```
apply -- [sd pre: (preemption(.driver\sum,
                               transaction(timeplus(.vhdltime,
                                                    vhdltime(5000000,0)),
                                           .a usxor .cin)))
          comod: (all)
            mod: (adder\pc,driver\sum)
           post: (#driver\sum
                    = inertial_update(
                        .driver\sum,
                        transaction(timeplus(.vhdltime,
                                             vhdltime(5000000,0)),
                                    .a usxor .cin)
                        ),
                  <VHDLTR>)]
```

<sdvs.5.11.19.2.3.1.2> *vhdl-signals*
signal-names[all]: *a, sum, cout*
simplify?[no]: *yes*

```
signal A :

  current value    =  a\86

  previous value   =  a\29

  projected output waveform =  ()

  driver history   =  (transaction(vhdltime(3000000,0),a\86),
                       transaction(vhdltime(0,0),a\29))


signal SUM :

  current value    =  sum\22

  previous value   =  sum\22

  projected output waveform =  (transaction(vhdltime(8000000,0),
                                            a\86 usxor cin\17))

  driver history   =  (transaction(vhdltime(0,0),sum\22))


signal COUT :
```

```
current value    =  cout\24

previous value   =  cout\24

projected output waveform =  (transaction(vhdltime(7000000,0),
                                          (x\13 && y\15 usor
                                           x\13 && cin\17) usor
                                           y\15 && cin\17))

driver history   =  (transaction(vhdltime(0,0),cout\24))
```

Observe how `transaction(vhdltime(5000000,0),a\29 usxor cin\17)` has been preempted from `driver\sum` by `transaction(vhdltime(8000000,0),a\86 usxor cin\17)`.

From this point on, the proof proceeds essentially as before, with the only difference being that the times to which `vhdltime` gets to advance are a little different. Note, in particular, that now `vhdltime(8000000,0)` is achieved.

```
<sdvs.5.11.19.2.3.1.2>  go
   until[]:  <CR>

        action -- <SUSPEND PROCESS: UPDATE_SUM>

        action -- <END EXECUTION CYCLE>

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc,vhdltime,vhdltime_previous,cout)
                   post: (#vhdltime = vhdltime(7000000,0),
                          #vhdltime_previous = .vhdltime,
                          <UPDATE SIGNALS>)]

        action -- <END EXECUTION CYCLE>

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>
```

```
        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc,vhdltime,vhdltime_previous,sum)
                   post: (#vhdltime = vhdltime(8000000,0),
                          #vhdltime_previous = .vhdltime,
                          <UPDATE SIGNALS>)]

        action -- <END EXECUTION CYCLE>

        action -- <BEGIN EXECUTION CYCLE:
                     1. ADVANCE EXECUTION TIME,
                     2. UPDATE SIGNALS,
                     3. RESUME PROCESSES>

        action -- <END VHDL MODEL EXECUTION>

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc)
                   post: (vhdl_model_execution_complete(adder))]

        go -- no more declarations or statements
```

<sdvs.5.11.19.2.3.1.13> *vhdltime*

```
        global time  =  8000000

        delta  time  =  0
```

<sdvs.5.11.19.2.3.1.13> *vhdl-signals*
signal-names[all]: *a, sum, cout*
simplify?[no]: *yes*

```
        signal A :

          current value     =  a\86

          previous value    =  a\29

          projected output waveform =  ()

          driver history    =  (transaction(vhdltime(3000000,0),a\86),
```

```
                              transaction(vhdltime(0,0),a\29))

        signal SUM :

          current value    =  a\86 usxor cin\17

          previous value   =  sum\22

          projected output waveform =  ()

          driver history   =  (transaction(vhdltime(8000000,0),
                                        a\86 usxor cin\17),
                               transaction(vhdltime(0,0),
                                        sum\22))


        signal COUT :

          current value    =  (x\13 && y\15 usor
                               x\13 && cin\17) usor
                               y\15 && cin\17

          previous value   =  cout\24

          projected output waveform =  ()

          driver history   =  (transaction(vhdltime(7000000,0),
                                        (x\13 && y\15 usor
                                         x\13 && cin\17) usor
                                         y\15 && cin\17),
                               transaction(vhdltime(0,0),
                                        cout\24))
```

<sdvs.5.11.19.2.3.1.13> *vhdl-processes*
process-names[all]: <*CR*>

```
        process UPDATE_A :

          current state      =  SUSPENDED


        process UPDATE_SUM :
```

```
          current state      =  SUSPENDED


        process UPDATE_COUT :

          current state      =  SUSPENDED


<sdvs.5.11.19.2.3.1.13>  whynotgoal
   simplify?[no]:  <CR>

 g(1) |#cout @ #sum| = |(x\36 ++ y\37) ++ cin\38|

<sdvs.5.11.19.2.3.1.13>  provebylemma
   formula to prove:  |.cout @ .sum| = |(.x ++ .y) ++ .cin|
       lemma name[]:  <CR>

          provebylemma append_cout_sum.lemma -- |.cout @ .sum|
                                                  = |(.x ++ .y) ++ .cin|

<sdvs.5.11.19.2.3.1.14>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.5.11.19.2.3.1.14>  close

          close -- 13 steps/applications

          open -- [sd pre: (~(preemption(.driver\sum,
                                         transaction(timeplus(.vhdltime,
                                                              vhdltime(5000000,0)),
                                                     .a usxor .cin))))
                    comod: (all)
                      mod: (all)
                     post: (|#cout @ #sum|
                               = |(x\36 ++ y\37) ++ cin\38|,
                            vhdl_model_execution_complete(adder))]

 Complete the proof.

<sdvs.5.11.19.2.3.2.1>  nsd

 [sd pre: (~(preemption(.driver\sum,
```

```
                        transaction(timeplus(.vhdltime,
                                             vhdltime(5000000,0)),
                                    .a usxor .cin))))
   comod: (all)
     mod: (adder\pc,driver\sum)
    post: (#driver\sum
             = inertial_update(.driver\sum,
                               transaction(timeplus(.vhdltime,
                                                    vhdltime(5000000,0)),
                                           .a usxor .cin)),
           <VHDLTR>)]

<sdvs.5.11.19.2.3.2.1>  apply
   sd/number[highest applicable/once]:

            apply -- [sd pre: (~(preemption(.driver\sum,
                                            transaction(timeplus(.vhdltime,
                                                                 vhdltime(5000000,0)),
                                                        .a usxor .cin))))
                      comod: (all)
                        mod: (adder\pc,driver\sum)
                       post: (#driver\sum
                                = inertial_update(
                                    .driver\sum,
                                    transaction(timeplus(.vhdltime,
                                                         vhdltime(5000000,0)),
                                                .a usxor .cin)
                                ),
                              <VHDLTR>)]

<sdvs.5.11.19.2.3.2.2>  vhdl-signals
   signal-names[all]: a, sum, cout
       simplify?[no]: yes


         signal A :

           current value    =  a\86

           previous value   =  a\29

           projected output waveform =  ()

           driver history   =  (transaction(vhdltime(3000000,0),a\86),
                                transaction(vhdltime(0,0),a\29))
```

```
signal SUM :

  current value    =  sum\22

  previous value   =  sum\22

  projected output waveform =  (transaction(vhdltime(5000000,0),
                                            a\86 usxor cin\17),
                                transaction(vhdltime(8000000,0),
                                            a\86 usxor cin\17))

  driver history   =  (transaction(vhdltime(0,0),sum\22))


signal COUT :

  current value    =  cout\24

  previous value   =  cout\24

  projected output waveform =  (transaction(vhdltime(7000000,0),
                                (x\13 && y\15 usor
                                 x\13 && cin\17) usor
                                 y\15 && cin\17))

  driver history   =  (transaction(vhdltime(0,0),cout\24))
```

We have entered the case where the value scheduled for sum at time vhdltime(5000000,0) will *not* be preempted; that is, the previously scheduled value a\29 usxor cin\17 and the newly scheduled value a\86 usxor cin\17 are considered equivalent, as reflected by examination of sum's projected output waveform after application of the state delta.

The proof now proceeds much as before, with the only difference being manifested, again, in the times to which vhdltime gets to advance.

```
<sdvs.5.11.19.2.3.2.2>  go
   until□:  <CR>

            action -- <SUSPEND PROCESS: UPDATE_SUM>

            action -- <END EXECUTION CYCLE>

            action -- <BEGIN EXECUTION CYCLE:
```

```
            1. ADVANCE EXECUTION TIME,
            2. UPDATE SIGNALS,
            3. RESUME PROCESSES>

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder\pc,vhdltime,vhdltime_previous,sum)
           post: (#vhdltime = vhdltime(5000000,0),
                  #vhdltime_previous = .vhdltime,
                  <UPDATE SIGNALS>)]

action -- <END EXECUTION CYCLE>

action -- <BEGIN EXECUTION CYCLE:
            1. ADVANCE EXECUTION TIME,
            2. UPDATE SIGNALS,
            3. RESUME PROCESSES>

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder\pc,vhdltime,vhdltime_previous,cout)
           post: (#vhdltime = vhdltime(7000000,0),
                  #vhdltime_previous = .vhdltime,
                  <UPDATE SIGNALS>)]

action -- <END EXECUTION CYCLE>

action -- <BEGIN EXECUTION CYCLE:
            1. ADVANCE EXECUTION TIME,
            2. UPDATE SIGNALS,
            3. RESUME PROCESSES>

apply -- [sd pre: (true)
          comod: (all)
            mod: (adder\pc,vhdltime,vhdltime_previous,sum)
           post: (#vhdltime = vhdltime(8000000,0),
                  #vhdltime_previous = .vhdltime,
                  <UPDATE SIGNALS>)]

action -- <END EXECUTION CYCLE>

action -- <BEGIN EXECUTION CYCLE:
            1. ADVANCE EXECUTION TIME,
            2. UPDATE SIGNALS,
            3. RESUME PROCESSES>
```

```
        action -- <END VHDL MODEL EXECUTION>

          *** Execution Time reached TIME'HIGH :
              no active drivers or resuming processes  ***

        apply -- [sd pre: (true)
                  comod: (all)
                    mod: (adder\pc)
                   post: (vhdl_model_execution_complete(adder))]

        go -- no more declarations or statements

<sdvs.5.11.19.2.3.2.16> vhdltime


        global time  =  8000000

        delta  time  =  0


<sdvs.5.11.19.2.3.2.16> vhdl-signals
   signal-names[all]: a, sum, cout
       simplify?[no]: yes


        signal A :

          current value     =  a\86

          previous value    =  a\29

          projected output waveform =  ()

          driver history    =  (transaction(vhdltime(3000000,0),a\86),
                                transaction(vhdltime(0,0),a\29))


        signal SUM :

          current value     =  a\86 usxor cin\17

          previous value    =  a\86 usxor cin\17

          projected output waveform =  ()
```

```
    driver history   =  (transaction(vhdltime(8000000,0),
                                     a\86 usxor cin\17),
                        transaction(vhdltime(5000000,0),
                                     a\86 usxor cin\17),
                        transaction(vhdltime(0,0),
                                     sum\22))


  signal COUT :

    current value    =  (x\13 && y\15 usor
                         x\13 && cin\17) usor
                         y\15 && cin\17

    previous value   =  cout\24

    projected output waveform =  ()

    driver history   =  (transaction(vhdltime(7000000,0),
                                     (x\13 && y\15 usor
                                      x\13 && cin\17) usor
                                      y\15 && cin\17),
                        transaction(vhdltime(0,0),
                                     cout\24))


<sdvs.5.11.19.2.3.2.16>  vhdl-processes
  process-names[all]:


  process UPDATE_A :

    current state      =  SUSPENDED


  process UPDATE_SUM :

    current state      =  SUSPENDED


  process UPDATE_COUT :

    current state      =  SUSPENDED
```

```
<sdvs.5.11.19.2.3.2.16>  whynotgoal
   simplify?[no]:

 g(1) |#cout @ #sum| = |(x\36 ++ y\37) ++ cin\38|

<sdvs.5.11.19.2.3.2.16> provebylemma
   formula to prove: |.cout @ .sum| = |(.x ++ .y) ++ .cin|
       lemma name[]:

            provebylemma append_cout_sum.lemma -- |.cout @ .sum|
                                                    = |(.x ++ .y) ++ .cin|

<sdvs.5.11.19.2.3.2.17>  whynotgoal
   simplify?[no]:

 The goal is TRUE.  Type 'close'.

<sdvs.5.11.19.2.3.2.17>  close

          close -- 16 steps/applications

        join -- [sd pre: (true)
                 comod: (all)
                   mod: (all)
                  post: (|#cout @ #sum|
                            = |(x\36 ++ y\37) ++ cin\38|,
                          vhdl_model_execution_complete(adder))]

      close -- 3 steps/applications

    join -- [sd pre: (true)
             comod: (all)
               mod: (all)
              post: (|#cout @ #sum| = |(x\36 ++ y\37) ++ cin\38|,
                     vhdl_model_execution_complete(adder))]

   close -- 19 steps/applications

 Complete the proof.

<sdvs.5.12>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.
```

<sdvs.5.12> *close*

close -- 11 steps/applications

<sdvs.6>

7.2.3 Batch proof

The batch proof shown here is essentially a dump of the proof developed in the preceding section, with successive **apply** commands merged into invocations of the go command. Recall that, with the **autoclose** flag set to **off**, the go command applies successive highest applicable state deltas until the top usable state delta is not applicable or the indicated condition (if any) is achieved.

```
(defproof full_adder_dataflow.proof
   "(setflag autoclose off,
     vhdltr \"testproofs/vhdl2/full_adder_dataflow.vhdl\",
     read \"testproofs/vhdl2/full_adder_dataflow.spec\",
     read \"testproofs/vhdl2/full_adder_dataflow.lemmas\",
     prove full_adder_dataflow.sd
       proof:
         (go vhdl_model_elaboration_complete(adder),
          prove g(2)
            proof:
              (go,
               cases .a = val(.driver\\a,.vhdltime_previous)
                 then proof:
                   (go,
                    provebylemma |.cout @ .sum|
                                  = |(.x ++ .y) ++ .cin|
                      using: append_cout_sum.lemma,
                    close)
                 else proof:
                   (go,
                    cases preemption(.driver\\sum,
                                     transaction(timeplus(.vhdltime,
                                                          vhdltime(5000000,0)),
                                                 .a usxor .cin))
                      then proof:
                        (go,
                         provebylemma |.cout @ .sum|
                                       = |(.x ++ .y) ++ .cin|
                           using: append_cout_sum.lemma,
                         close)
                      else proof:
                        (go,
                         provebylemma |.cout @ .sum|
                                       = |(.x ++ .y) ++ .cin|
                           using: append_cout_sum.lemma,
                         close))),
          close))")
```

7.2.4 Lemma

Here we record the lemma used in the above proof. It has a straightforward proof by exhaustive case analysis.

```
(deflemma append_cout_sum.lemma
   "(((((lh(x) = 1 &
         lh(y) = 1) &
           lh(cin) = 1) &
             lh(sum) = 1) &
               lh(cout) = 1) &
                 sum = (x usxor y) usxor cin) &
                   cout = (x && y usor x && cin) usor y && cin
     --> |cout @ sum| = |(x ++ y) ++ cin|"
    (x y cin sum cout) nil nil nil
    :proof "(provelemma append_cout_sum.lemma
              proof:
                mcases
                  (case: (x = 0(1) & y = 0(1)) & cin = 0(1)
                     proof: close
                   case: (x = 0(1) & y = 0(1)) & cin = 1(1)
                     proof: close
                   case: (x = 0(1) & y = 1(1)) & cin = 0(1)
                     proof: close
                   case: (x = 0(1) & y = 1(1)) & cin = 1(1)
                     proof: close
                   case: (x = 1(1) & y = 0(1)) & cin = 0(1)
                     proof: close
                   case: (x = 1(1) & y = 0(1)) & cin = 1(1)
                     proof: close
                   case: (x = 1(1) & y = 1(1)) & cin = 0(1)
                     proof: close
                   case: (x = 1(1) & y = 1(1)) & cin = 1(1)
                     proof: close))")
```

7.3 ISPS

7.3.1 TR: Translator from ISPS to state deltas

This section describes the action of the TR translator on the machine description language ISPS.

In fact, there are two different versions of the translator from ISPS to state deltas. The new translator will be discussed only in the last section of this chapter. It is still to be considered experimental, although it will eventually replace the old translator. It has been generated by the same uniform method as the translators for Ada and VHDL, and recognizes a slightly larger piece of ISPS (it allows "don't care" digits, and bit order in bitstrings can be low to high).

The version of ISPS that the (old) translator (TR) recognizes differs from the version described in the ISPS manual [4] in several respects. The first category of differences contains those aspects of the "official" ISPS that TR does not support; these include parallelism and two's-complement arithmetic.

The second category of differences consists of extra features that SDVS needs for the implementation proof paradigm. For example, when one is not interested in implementing the action of all target places, some of the machine variables ("place" names) must be designated as significant and the others as auxiliary. The mapping is defined only on the designated significant places. Another useful feature is the capability to intersperse standard ISPS code with state deltas. This can be used when one is not interested in the details of how a certain postcondition was brought about, but only in its effect, or in case that effect is not expressible in ISPS.

The semantics of TR are described in [17], [20], and [21]; problems with ISPS are described in [22].

7.3.2 Marking

SDVS does the processing necessary to turn an ISPS program into an equivalent state delta or set of state deltas. Thus, ISPS programs can be used in, or as, preconditions or postconditions of state deltas.

We present an example illustrating the capability to execute from an ISPS mark point. One can run a set of example ISPS proofs by typing *eval (runtestproofs *isps-tests*)*.

When dealing with a proof based on state deltas created by TR from an ISPS program, the user does not have a convenient method of handling the specific state deltas representing the "continuation" of the program from each control point. To solve that problem, the system allows the user to label the location of control points in the ISPS program.

The initial and final control points are named by the system <machine-name>\STARTED and <machine-name>\HALTED, respectively. The exit point for an internal subroutine,

<subroutine>, is <subroutine>\exited.

Consider the following ISPS program, gcd.isp:

```
gcd.machine US := BEGIN  ! gcd algorithm computes gcd(x,y)
                         ! for inputs x and y
** local.variables **

x<15:0>,          ! input variable x
y<15:0>,          ! input variable y
twos<5:0>,        ! indicates common factor of twos between x and y
gcdresult<15:0>   ! result of gcd(x,y)

** algorithm **

gcd MAIN :=  BEGIN
  twos _ LAST.ONE(x OR y)  NEXT       ! store common factor of twos
  y _ y SRO LAST.ONE(y) NEXT          ! strip low-order zeros from y
  x _ x SRO LAST.ONE(x) NEXT          ! strip low-order zeros from x
  REPEAT                              ! main loop
      BEGIN
    m1:=  IF x LSS y => x@y _ y@x  NEXT      ! swap x,y if x<y
      x _ x - y  NEXT                        ! assign x-y to x
    m2:=  IF x EQL 0 =>                      ! if x=0 (finished) then
        (m4 := gcdresult _ y NEXT            !   assign y to gcdxy,
         gcdresult _ gcdresult SL0 twos NEXT !   remember common twos,
         LEAVE gcd)   NEXT !   and exit
     m3:= x _ x SRO LAST.ONE(x)              ! strip low-order zeros from x
    END
END
END
```

The command **mpisps** generates state deltas corresponding to the state changes between mark points, instead of every state change represented in the unmarked ISPS program. If **mpisps** is used on an ISPS program with a potentially infinite loop in which the loop does not have a mark point at the top, **mpisps** will not terminate. Gcd.isp has five mark points, including the initial state, which is a default mark point.

Mpisps prompts for starting mark point, stopping mark point, and preconditions.

```
<sdvs.1>  mpisps
   path name[testproofs/alias.isp]:  testproofs/gcd.isp
            starting mark point[]:  <CR>
             ending mark points[]:  <CR>
                  preconditions[]:  <CR>
             unique name level[1]:  <CR>
```

```
Parsing ISPS file -- "testproofs/gcd.isp"

Markpoint-to-markpoint translating ISPS file -- "testproofs/gcd.isp"

[sd pre: (.gcd.machine\upc = gcd.machine\started)
    mod: (x,twos,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>,
          #y = (zeros(|lastone(.y)|) @ .y)
                <15 + |lastone(.y)|:|lastone(.y)|>,
          #twos = lastone(.x usor .y))]

[sd pre: (|.y| gt |.x|,.gcd.machine\upc = m1)
    mod: (x,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.y -- .x)<15:0>,#y = .x)]

[sd pre: (|.y| le |.x|,.gcd.machine\upc = m1)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.x -- .y)<15:0>)]

[sd pre: (|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]

[sd pre: (|.x| ~= 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m3)]

[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]

[sd pre: (.gcd.machine\upc = m3)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>)]
```

The flag **displaympsds** was on. If it were off, the above state deltas would not be displayed.

```
<sdvs.2>  ppsd
   state delta:  mpisps
```

```
              file name:  gcd.isp
   starting mark point[]:  <CR>
    ending mark points[]:  <CR>
         preconditions[]:  <CR>

covering(gcd.machine,x,y,twos,gcdresult,gcd.machine\upc)
declare(x,type(bitstring,16))
declare(y,type(bitstring,16))
declare(twos,type(bitstring,6))
declare(gcdresult,type(bitstring,16))
[sd pre: (.gcd.machine\upc = gcd.machine\started)
    mod: (x,twos,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>,
          #y = (zeros(|lastone(.y)|) @ .y)
                <15 + |lastone(.y)|:|lastone(.y)|>,
          #twos = lastone(.x usor .y))]
[sd pre: (|.y| gt |.x|,.gcd.machine\upc = m1)
    mod: (x,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.y -- .x)<15:0>,#y = .x)]
[sd pre: (|.y| le |.x|,.gcd.machine\upc = m1)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.x -- .y)<15:0>)]
[sd pre: (|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]
[sd pre: (|.x| ~= 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m3)]
[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]
[sd pre: (.gcd.machine\upc = m3)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>)]
```

Now we will use **mpisps** with mark points chosen.

```
<sdvs.2>  mpisps
   path name[testproofs/gcd.isp]:  testproofs/gcd.isp
          starting mark point[]:  m2
```

```
            ending mark points[]:  m3
                 preconditions[]:  <CR>
            unique name level[1]:  <CR>

Parsing ISPS file -- "testproofs/gcd.isp"

Markpoint-to-markpoint translating ISPS file -- "testproofs/gcd.isp"

[sd pre: (|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]

[sd pre: (|.x| ~= 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m3)]

[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]

<sdvs.3>  mpisps
  path name[testproofs/gcd.isp]:  <CR>
          starting mark point[]:  m2
           ending mark points[]:  <CR>
                preconditions[]:  <CR>
           unique name level[1]:  <CR>

Parsing ISPS file -- "testproofs/gcd.isp"

Markpoint-to-markpoint translating ISPS file -- "testproofs/gcd.isp"

[sd pre: (|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]

[sd pre: (|.x| ~= 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m3)]

[sd pre: (.gcd.machine\upc = m3)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>)]
```

```
[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]

[sd pre: (|.y| le |.x|,.gcd.machine\upc = m1)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.x -- .y)<15:0>)]

[sd pre: (|.y| gt |.x|,.gcd.machine\upc = m1)
    mod: (x,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.y -- .x)<15:0>,#y = .x)]

<sdvs.4>  mpisps
  path name[testproofs/gcd.isp]:  <CR>
         starting mark point[]:  m2
          ending mark points[]:  <CR>
               preconditions[]:  |.x| ge |.y|
          unique name level[1]:  <CR>

Parsing ISPS file -- "testproofs/gcd.isp"

Markpoint-to-markpoint translating ISPS file -- "testproofs/gcd.isp"

[sd pre: (|.x| ge |.y|,|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]

[sd pre: (|.x| ge |.y|,|.x| ~= 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m3)]

[sd pre: (.gcd.machine\upc = m3)
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m1,
          #x = (zeros(|lastone(.x)|) @ .x)
                <15 + |lastone(.x)|:|lastone(.x)|>)]

[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]

[sd pre: (|.y| le |.x|,.gcd.machine\upc = m1)
```

```
    mod: (x,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.x -- .y)<15:0>)]

[sd pre: (|.y| gt |.x|,.gcd.machine\upc = m1)
    mod: (x,y,gcd.machine\upc)
   post: (#gcd.machine\upc = m2,#x = (.y -- .x)<15:0>,#y = .x)]
```

```
<sdvs.5>  mpisps
  path name[testproofs/gcd.isp]:  <CR>
         starting mark point[]:  m2
          ending mark points[]:  <CR>
               preconditions[]:  |.x| = 0
          unique name level[1]:  <CR>

Parsing ISPS file -- "testproofs/gcd.isp"

Markpoint-to-markpoint translating ISPS file -- "testproofs/gcd.isp"

[sd pre: (|.x| = 0,.gcd.machine\upc = m2)
    mod: (gcd.machine\upc)
   post: (#gcd.machine\upc = m4)]

[sd pre: (.gcd.machine\upc = m4)
    mod: (gcdresult,gcd.machine\upc)
   post: (#gcd.machine\upc = gcd.machine\halted,
          #gcdresult = (.y @ zeros(|.twos|))<15:0>)]
```

The differences between **isps** and **mpisps** are as follows:

1. **isps** gives an incremental translation (with TRs in the postcondition); **mpisps** gives a set of state deltas;
2. **isps** translates every state ISPS state change; **mpisps** accumulates effects from mark point to mark point;
3. **mpisps** takes account of extensions of ISPS by state deltas, assumptions, and external and auxiliary variables; and
4. **isps**(file.isp) should be used only in the precondition of a state delta (as a host description).

7.3.3 Extensions of ISPS

The user may extend ISPS code in two main ways:

1. by interspersing assumptions or state deltas between ISPS statements, and
2. by declaring some ISPS variables to be external or auxiliary.

These extensions were found to be useful in specifying real machines in the context of setting up implementation proofs.

7.3.4 Extending ISPS by assumptions and state deltas

The two methods for extending ISPS that are discussed in this section are

1. assumptions *!![ASSUME: (expr)]*, and
2. inserting state deltas *!![SD (pre) (comod) (mod) (post)]*.

The expr field in *assumption* is any state delta formula (note that a statement such as "#x = 1" is not a legal state delta formula); it is interpreted to be a precondition to the rest of the ISPS routine. In other words, if the assumption is not true, execution cannot continue from that point.

The extended state delta is interpreted with the same internal semantics as any state delta, and with the same control as if it had been a regular ISPS statement. It is useful for expressing state changes that cannot be expressed in ISPS. Notice that one may make a static assertion by using an extended state delta with nil precondition and nil mod list.

As an example, consider the following extended ISPS program `extest2.isp`:

```
sd.machine US :=
BEGIN
**Registers**

x<15:0>, y<15:0>

**Algorithm**

exec MAIN:=
BEGIN

!![EXTSD: () (|.x| ge |.y|) () (x, y) (#x = 0(16) or #y = 0(16))] NEXT
POINT:=
if x eql 0 => y _ 1 NEXT
if y eql 0 => x _ 0
END
END
```

Let us **mpisps** it and look at the resulting state deltas.

```
<sdvs.1>  mpisps
   path name[testproofs/gcd.isp]:  testproofs/extest2.isp
           starting mark point[]:  <CR>
            ending mark points[]:  <CR>
                 preconditions[]:  <CR>
            unique name level[1]:  <CR>

 Parsing ISPS file -- "testproofs/extest2.isp"

 Markpoint-to-markpoint translating ISPS file -- "testproofs/extest2.isp"

 [sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
     mod: (y,x,sd.machine\upc)
    post: (#x = 0(16) or #y = 0(16),#sd.machine\upc = point)]

 [sd pre: (|.x| lt |.y|,.sd.machine\upc = sd.machine\started)
     mod: (sd.machine\upc)
    post: (#sd.machine\upc = point)]

 [sd pre: (|.x| = 0,.sd.machine\upc = point)
     mod: (y,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]

 [sd pre: (|.x| ~= 0 & .sd.machine\upc = point,|.y| = 0)
     mod: (x,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#x = 0(16))]

 [sd pre: (|.x| ~= 0 & .sd.machine\upc = point,|.y| ~= 0)
     mod: (sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted)]

<sdvs.2>  ppsd
   state delta:  mpisps
                 file name:  extest2.isp
    starting mark point[]:  <CR>
     ending mark points[]:  <CR>
          preconditions[]:  <CR>

 covering(sd.machine,x,y,sd.machine\upc)
 declare(x,type(bitstring,16))
 declare(y,type(bitstring,16))
 [sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
     mod: (y,x,sd.machine\upc)
```

```
    post: (#x = 0(16) or #y = 0(16),#sd.machine\upc = point)]
 [sd pre: (|.x| lt |.y|,.sd.machine\upc = sd.machine\started)
    mod: (sd.machine\upc)
    post: (#sd.machine\upc = point)]
 [sd pre: (|.x| = 0,.sd.machine\upc = point)
    mod: (y,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]
 [sd pre: (|.x| ~= 0 & .sd.machine\upc = point,|.y| = 0)
    mod: (x,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#x = 0(16))]
 [sd pre: (|.x| ~= 0 & .sd.machine\upc = point,|.y| ~= 0)
    mod: (sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted)]
```

Let extest.isp be the above without POINT:

```
sd.machine US :=
BEGIN
**Registers**

x<15:0>, y<15:0>

**Algorithm**

exec MAIN:=
BEGIN

!![EXTSD: () (|.x| ge |.y|) () (x, y) (#x = 0(16) or #y = 0(16))] NEXT

if x eql 0 => y _ 1 NEXT
if y eql 0 => x _ 0
END
END
```

```
<sdvs.1>  mpisps
   path name[testproofs/extest2.isp]:  testproofs/extest.isp
              starting mark point[]:  <CR>
               ending mark points[]:  <CR>
                    preconditions[]:  <CR>
               unique name level[1]:  <CR>

 Parsing ISPS file -- "testproofs/extest.isp"

 Markpoint-to-markpoint translating ISPS file -- "testproofs/extest.isp"
```

```
[sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
    mod: (x,y,sd.machine\upc)
   post: (exists gv-y-11054 exists gv-x-11053 (((gv-x-11053 = 0(16) or
                                                  gv-y-11054 = 0(16)) &
                                                lh(gv-x-11053) = 16 &
                                                lh(gv-y-11054) = 16) &
                                               (|gv-x-11053| = 0
                                                  --> #sd.machine\upc
                                                        = sd.machine\halted &
                                                      #y = 0(14) @
                                                            1(2) &
                                                      #x = 0(16))))]

[sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
    mod: (x,y,sd.machine\upc)
   post: (exists gv-y-11054 exists gv-x-11053 (((gv-x-11053 = 0(16) or
                                                  gv-y-11054 = 0(16)) &
                                                lh(gv-x-11053) = 16 &
                                                lh(gv-y-11054) = 16) &
                                               (|gv-x-11053| ~= 0
                                                  --> #sd.machine\upc
                                                        = sd.machine\halted &
                                                      #x = 0(16) &
                                                      #y = 0(16))))]

[sd pre: (|.x| lt |.y| & .sd.machine\upc = sd.machine\started,|.x| = 0)
    mod: (y,sd.machine\upc)
   post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]

[sd pre: (|.x| lt |.y| & .sd.machine\upc = sd.machine\started,
          |.x| ~= 0)
    mod: (sd.machine\upc)
   post: (#sd.machine\upc = sd.machine\halted)]

<sdvs.2>  ppsd
  state delta:  mpisps
              file name:  extest.isp
   starting mark point[]:  <CR>
    ending mark points[]:  <CR>
         preconditions[]:  <CR>

covering(sd.machine,x,y,sd.machine\upc)
declare(x,type(bitstring,16))
declare(y,type(bitstring,16))
[sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
```

```
     mod: (x,y,sd.machine\upc)
    post: (exists gv-y-11054 exists gv-x-11053 (((gv-x-11053 = 0(16) or
                                                   gv-y-11054 = 0(16)) &
                                                  lh(gv-x-11053) = 16 &
                                                  lh(gv-y-11054) = 16) &
                                                 (|gv-x-11053| = 0
                                                    --> #sd.machine\upc
                                                          = sd.machine\halted &
                                                        #y = 0(14) @
                                                              1(2) &
                                                        #x = 0(16))))]
  [sd pre: (|.x| ge |.y|,.sd.machine\upc = sd.machine\started)
     mod: (x,y,sd.machine\upc)
    post: (exists gv-y-11054 exists gv-x-11053 (((gv-x-11053 = 0(16) or
                                                   gv-y-11054 = 0(16)) &
                                                  lh(gv-x-11053) = 16 &
                                                  lh(gv-y-11054) = 16) &
                                                 (|gv-x-11053| ~= 0
                                                    --> #sd.machine\upc
                                                          = sd.machine\halted &
                                                        #x = 0(16) &
                                                        #y = 0(16))))]
  [sd pre: (|.x| lt |.y| & .sd.machine\upc = sd.machine\started,|.x| = 0)
     mod: (y,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]
  [sd pre: (|.x| lt |.y| & .sd.machine\upc = sd.machine\started,
            |.x| ~= 0)
     mod: (sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted)]
```

It is clear that the following state delta (call it extsd1) is true:

```
  [sd pre: (mpisps(extest2.isp),.sd.machine\upc = sd.machine\started)
     mod: (all)
    post: (|#x| le |#y|,#sd.machine\upc = sd.machine\halted)]
```

and the following proof works:

```
  (prove extsd1
     proof:
       cases |.x| ge |.y|
         then proof:
           (apply,
            cases |.x| = 0
              then proof:
```

```
            (apply,
             close)
          else proof:
            (notice |.y| = 0,
             apply,
             close))
      else proof:
        (apply,
         cases |.x| = 0
           then proof:
             (apply,
              close)
           else proof:
             cases |.y| = 0
               then proof:
               else proof:
                 (apply,
                  close)))
```

As a good exercise, try to input the above state delta and proof in the editor, using the **defsd** and **defproof** functions. Remember to use two slashes "\\" in the editor to get one real slash.

We cannot currently prove the corresponding state delta involving `extest.isp`; any state deltas resulting from **mpisps** that contain existential quantifiers should be suspect. The user should eliminate these quantifiers by adding mark points in suitable places in the original ISPS.

Now let us examine the state delta formed by making $.x \geq .y$ an assumption. Call the following extended ISPS program `extest3.isp`:

```
sd.machine US :=
BEGIN
**Registers**

x<15:0>, y<15:0>

**Algorithm**

exec MAIN:=
BEGIN

!![ASSUME: (|.x| ge |.y|)] NEXT
if x eql 0 => y _ 1 NEXT
if y eql 0 => x _ 0
END
```

```
  END
```

```
<sdvs.1>  mpisps
   path name[testproofs/extest.isp]:  testproofs/extest3.isp
              starting mark point[]:  <CR>
               ending mark points[]:  <CR>
                    preconditions[]:  <CR>
               unique name level[1]:  <CR>

 Parsing ISPS file -- "testproofs/extest3.isp"

 Markpoint-to-markpoint translating ISPS file -- "testproofs/extest3.isp"

 [sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,|.x| = 0)
     mod: (y,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]

 [sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,
           |.x| ~= 0,|.y| = 0)
     mod: (x,sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted,#x = 0(16))]

 [sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,
           |.x| ~= 0,|.y| ~= 0)
     mod: (sd.machine\upc)
    post: (#sd.machine\upc = sd.machine\halted)]

<sdvs.2>  ppsd
   state delta:  mpisps
                 file name:  extest3.isp
    starting mark point[]:  <CR>
     ending mark points[]:  <CR>
          preconditions[]:  <CR>

 covering(sd.machine,x,y,sd.machine\upc)
 declare(x,type(bitstring,16))
 declare(y,type(bitstring,16))
 [sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,|.x| = 0)
     mod: (y,sd.machine\upc)
```

```
   post: (#sd.machine\upc = sd.machine\halted,#y = 0(14) @ 1(2))]
[sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,
          |.x| ~= 0,|.y| = 0)
    mod: (x,sd.machine\upc)
   post: (#sd.machine\upc = sd.machine\halted,#x = 0(16))]
[sd pre: (|.x| ge |.y| & .sd.machine\upc = sd.machine\started,
          |.x| ~= 0,|.y| ~= 0)
    mod: (sd.machine\upc)
   post: (#sd.machine\upc = sd.machine\halted)]
```

7.3.5 External and auxiliary variables

External and auxiliary variables are introduced into ISPS descriptions in order to extend the possibilities of expression, not just to facilitate expression. These extended possibilities are reflected in the translation of the description into state deltas and the methods of proof needed to verify claims of implementation between two levels of description.

Both external and auxiliary variables satisfy specification needs arising from real problems. External variables have their intuitive motivation in "input variables," that is, variables whose value may change at random, upon receipt of a signal from some external source (external with respect to the level of description in which they appear designated as "external"), in addition to any changes explicitly required by that description.

The idea for auxiliary variables is found in the concept of temporary variables. Generally speaking, the designation "auxiliary" is used for any variable whose contents are not to be relied on, or even considered, by any "outside" observer (although of course they may be essential to the internal workings of the description). When viewed from the outside, auxiliary variables are not considered to be part of the state of the system.

7.3.6 External variables

The suffix *!!ext* may be appended to any ISPS declaration, e.g.

X<15:0>!!ext.

This indicates that the variable may change value during any state change explicitly allowed by the ISPS program. There is no need to change the syntax or semantics of state deltas to account for the external variables. An ISPS program with *ext* is translated into state deltas just as before, with the addition that the external variables appear in every mod list.

In the case of markpoint-to-markpoint translation, care must be taken, for example, when there is a case split on an external variable between the starting and ending markpoint. However, when we take the view that markpoint-to-markpoint translation equals the composition of the state deltas representing the translation of the fine-grained state changes, the problem of external variables is just a subcase of the general problem (remember that the only special handling that external variables need is to be placed in every mod list).

For example, consider the machine (on file extest4.isp):

```
sd.machine US :=
BEGIN
**Registers**

x<15:0>,
y<15:0>!!ext

**Algorithm**

exec MAIN:=
BEGIN

if x eql 0 => y _ 1 NEXT
if y eql 0 => x _ 0
END
END
```

and consider the state delta

```
<sdvs.1>  ppsd
   state delta:  extsd

 [sd pre: (|.x| = 1,isps(extest4.isp),
           .sd.machine\upc = sd.machine\started)
     mod: (all)
    post: (#sd.machine\upc = sd.machine\halted,|#x| = 0 or |#x| = 1)]
```

The following proof works:

```
<sdvs.1>  pp
   object:  extproof

 proof extproof:

  prove extsd
    proof:
      (apply,
       cases |.y| = 0
         then proof:
           (apply 3,
            close)
         else proof:
```

```
          (apply 2,
           close))

<sdvs.1>  interpret
   proof name:  extproof

 open -- [sd pre: (|.x| = 1,isps(extest4.isp),
                   .sd.machine\upc = sd.machine\started)
             mod: (all)
            post: (#sd.machine\upc = sd.machine\halted,
                   |#x| = 0 or |#x| = 1)]

   apply -- [sd pre: (.sd.machine\upc = sd.machine\started,
                      .x == 0(2) ~= 1(1))
               mod: (sd.machine\upc)
              post: ([tr in SD.MACHINE IF;])]

   cases -- |.y| = 0

     open -- [sd pre: (|.y| = 0)
               comod: (all)
                 mod: (all)
                post: (#sd.machine\upc = sd.machine\halted,
                       |#x| = 0 or |#x| = 1)]

       apply -- [sd pre: (.y == 0(2) = 1(1))
                  comod: (sd.machine\upc)
                    mod: (sd.machine\upc)
                   post: ([tr in SD.MACHINE X_...;])]

       apply -- [sd pre: (true)
                  comod: (sd.machine\upc)
                    mod: (sd.machine\upc,x)
                   post: (#x = 0(14) @ 0(2),
                          [tr @SD.MACHINE\halted])]

       apply -- [sd pre: (true)
                  comod: (sd.machine\upc)
                    mod: (sd.machine\upc)
                   post: (#sd.machine\upc = sd.machine\halted)]

     close -- 3 steps/applications

     open -- [sd pre: (~(|.y| = 0))
               comod: (all)
```

```
                mod: (all)
               post: (#sd.machine\upc = sd.machine\halted,
                      |#x| = 0 or |#x| = 1)]

      apply -- [sd pre: (.y == 0(2) ~= 1(1))
                comod: (sd.machine\upc)
                  mod: (sd.machine\upc)
                 post: ([tr @SD.MACHINE\halted])]

      apply -- [sd pre: (true)
                comod: (sd.machine\upc)
                  mod: (sd.machine\upc)
                 post: (#sd.machine\upc = sd.machine\halted)]

    close -- 2 steps/applications

  join -- [sd pre: (true)
           comod: (all)
             mod: (all)
            post: (#sd.machine\upc = sd.machine\halted,
                   |#x| = 0 or |#x| = 1)]

close -- 2 steps/applications
```

7.3.7 Auxiliary variables

The suffix *!!aux* may be appended to any ISPS declaration, e.g.

X<15:0>!!aux.

The difference between the semantics of such an annotated ISPS program and the semantics of an unannotated one becomes apparent only when one considers the interaction of the programs with another level. Auxiliary variables in target or host cannot play a role in the mapping. Thus, target auxiliary variables are not mapped from, and host auxiliary variables are not mapped to. Auxiliary variables do not appear in state deltas that are the result of **mpisps**.

Consider the machine

```
aux.machine US :=
BEGIN
**Registers**

x<15:0>,
y<15:0>,
temp<15:0>!!aux
```

```
**Algorithm**

exec MAIN:=
BEGIN
temp _ x next
x _ y next
y _ temp
END
END
```

```
<sdvs.1> ppsd
   state delta: mpisps
                file name: auxtest.isp
    starting mark point[]: <CR>
     ending mark points[]: <CR>
          preconditions[]: <CR>

covering(aux.machine,x,y,aux.machine\upc)
declare(x,type(bitstring,16))
declare(y,type(bitstring,16))
[sd pre: (.aux.machine\upc = aux.machine\started)
    mod: (y,x,aux.machine\upc)
   post: (#aux.machine\upc = aux.machine\halted,#y = .x,#x = .y)]
```

Now we shall construct a theorem saying that **auxtest** implements itself.

```
<sdvs.1> implementation
        theorem name: aux.thm
    upper-level spec: mpisps
                file name: auxtest.isp
    starting mark point[]: <CR>
     ending mark points[]: <CR>
          preconditions[]: <CR>
    lower-level spec: isps
           file name: auxtest.isp
            mappings: mapping(.x, .x), mapping(.y, .y),
                mapping(.aux.machine\upc,.aux.machine\upc)
         constants[]: <CR>
        invariants[]: <CR>

 Implementation theorem 'aux.thm' created.

<sdvs.1> ppsd
   state delta: aux.thm
```

```
 [sd pre: (isps(auxtest.isp),
          aux.thm.places = union(x,y,aux.machine\upc,
          aux.machine\aux),
          aux.thm.mapped.places = union(x,y,aux.machine\upc),
          aux.thm.unmapped.places
             = diff(aux.thm.places,aux.thm.mapped.places))
    post: (alldisjoint(x,y,aux.machine\upc),
          [sd pre: (true)
            comod: (all)
             post: (forall a1 (lh(a1) = 16 --> lh(a1) = 16),
                    forall a1 (lh(a1) = 16 --> lh(a1) = 16))],
          [sd pre: (.aux.machine\upc = aux.machine\started)
              mod: (y,x,aux.machine\upc,aux.thm.unmapped.places)
             post: (#aux.machine\upc = aux.machine\halted,#y = .x,
                    #x = .y)])]

<sdvs.1>  prove
   state delta[]:  aux.thm
   proof[]:  <CR>

 open -- [sd pre: (isps(auxtest.isp),
                   aux.thm.places
                      = union(x,y,aux.machine\upc,aux.machine\aux),
                   aux.thm.mapped.places = union(x,y,aux.machine\upc),
                   aux.thm.unmapped.places
                      = diff(aux.thm.places,aux.thm.mapped.places))
             post: (alldisjoint(x,y,aux.machine\upc),
                   [sd pre: (true)
                     comod: (all)
                      post: (forall a1 (lh(a1) = 16 --> lh(a1) = 16),
                             forall a1 (lh(a1) = 16 --> lh(a1) = 16))],
                   [sd pre: (.aux.machine\upc = aux.machine\started)
                       mod: (y,x,aux.machine\upc,aux.thm.unmapped.places)
                      post: (#aux.machine\upc = aux.machine\halted,
                             #y = .x,#x = .y)])]

 Complete the proof.

<sdvs.1.1>  whynotgoal
   simplify?[no]:  <CR>

 g(2) [sd pre: (true)
       comod: (all)
        post: (forall a1 (lh(a1) = 16 --> lh(a1) = 16),
```

```
                forall a1 (lh(a1) = 16 --> lh(a1) = 16))]
 g(3) [sd pre: (.aux.machine\upc = aux.machine\started)
         mod: (y,x,aux.machine\upc,aux.thm.unmapped.places)
        post: (#aux.machine\upc = aux.machine\halted,#y = .x,#x = .y)]

<sdvs.1.1>  prove
   state delta[]:  g
          number:  2
   proof[]:  <CR>

   open -- [sd pre: (true)
             comod: (all)
              post: (forall a1 (lh(a1) = 16 --> lh(a1) = 16),
                     forall a1 (lh(a1) = 16 --> lh(a1) = 16))]

   close -- 0 steps/applications

 Complete the proof.

<sdvs.1.2>  prove
   state delta[]:  g
          number:  3
   proof[]:  <CR>

   open -- [sd pre: (.aux.machine\upc = aux.machine\started)
              mod: (y,x,aux.machine\upc,aux.thm.unmapped.places)
             post: (#aux.machine\upc = aux.machine\halted,#y = .x,
                    #x = .y)]

 Complete the proof.

<sdvs.1.2.1>  *

     apply -- [sd pre: (.aux.machine\upc = aux.machine\started)
                 mod: (aux.machine\upc,temp)
                post: (#temp = .x,
                       [tr in AUX.MACHINE X_...; Y_...;])]

     apply -- [sd pre: (true)
                comod: (aux.machine\upc)
                  mod: (aux.machine\upc,x)
                 post: (#x = .y,
                        [tr in AUX.MACHINE Y_...;])]

     apply -- [sd pre: (true)
```

```
                comod: (aux.machine\upc)
                  mod: (aux.machine\upc,y)
                 post: (#y = .temp,
                        [tr @AUX.MACHINE\halted])]

    apply -- [sd pre: (true)
                comod: (aux.machine\upc)
                  mod: (aux.machine\upc)
                 post: (#aux.machine\upc = aux.machine\halted)]

  close -- 4 steps/applications

close -- 2 steps/applications
```

7.3.8 The new ISPS translator

The new translator can be accessed by the command **ispstr**. The associated predicate is **newisps**. We present an example comparing the new with the old translator on the ISPS program `inc1.isp`:

```
! inc1.ISP

inc1 US := (

**Registers**

x<7:0>

**Processes**

inc1 MAIN := BEGIN

 REPEAT  BEGIN
 loop1:=          x _ x + 1
         END
END
)
```

First, using the new translator:

```
<sdvs.1>  pp
   object:  newinc0.sd
```

```
[sd pre: (newisps(inc1.isp))
   post: (newisps(inc1.isp))]
```

We would expect this to be true and trivially provable, and it is with the new translator.

```
<sdvs.1>  setflag
   flag variable:  autoclose
   on or off[off]:  off

 setflag autoclose -- off

<sdvs.2>  prove
   state delta[]:  newinc0.sd
   proof[]:  <CR>

 open -- [sd pre: (newisps(inc1.isp))
            post: (newisps(inc1.isp))]

 Complete the proof.

<sdvs.2.1>  goals

 g(1) covering(inc1,inc1\upc,x)
 g(2) declare(x,type(bitstring,8))
 g(3) [sd pre: (.inc1\upc = inc1\started)
        comod: (all)
          mod: (inc1\upc)
         post: ([ispstr t(inc1) inc1 ...])]

<sdvs.2.1>  whynotgoal
   simplify?[no]:  <CR>

 The goal is TRUE.  Type 'close'.

<sdvs.2.1>  close

 close -- 0 steps/applications

<sdvs.3>  setflag
   flag variable:  autoclose
   on or off[on]:  on

 setflag autoclose -- on
```

Using the old translator things are not so trivial:

```
<sdvs.1>  pp
   object:  newinc1.sd

 [sd pre: (isps(inc1.isp))
    post: (isps(inc1.isp))]

<sdvs.1>  prove
   state delta[]:  newinc1.sd
   proof[]:  <CR>

 open -- [sd pre: (isps(inc1.isp))
           post: (isps(inc1.isp))]

 Complete the proof.

<sdvs.1.1>  whynotgoal
   simplify?[no]:  <CR>

 g(3) [tr @INC1\STARTED in INC1 REPEAT;]
 g(4) [tr @LOOP1 in INC1 X_...; REPEAT;]
```

In fact, it appears that this is unprovable in SDVS 11.

Index

References

[1] L. G. Marcus, "SDVS 11 Users' Manual," Technical Report ATR-92(2778)-8, The Aerospace Corporation, September 1992.

[2] T. K. Menas, "The Relation of the Temporal Logic of the State Delta Verification System (SDVS) to Classical First-Order Temporal Logic," Technical Report ATR-90(5778)-10, The Aerospace Corporation, September 1990.

[3] U. S. Department of Defense, *Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A)*, 22 January 1983.

[4] M. R. Barbacci, G. E. Barnes, R. G. Cattell, and D. P. Siewiorek, "The ISPS Computer Description Language," Technical Report CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, August 1979.

[5] IEEE, *Standard VHDL Language Reference Manual*, 1988. IEEE Std. 1076-1987.

[6] M. M. Cutler, "Verifying Implementation Correctness Using the State Delta Verification System (SDVS)," in *Proceedings of the 11th National Computer Security Conference*, National Bureau of Standards/National Computer Security Center, October 1988.

[7] T. Redmond, "Simplifier Description," Technical Report ATR-86A(8554)-2, The Aerospace Corporation, September 1987.

[8] J. V. Cook and J. Doner, "User Defined Data Types in the State Delta Verification System (SDVS)," Technical Report TR-0090(5920-07)-1, The Aerospace Corporation, September 1990.

[9] R. S. Boyer and J. S. Moore, *The User's Manual for a Computational Logic.* Computation Logic, Inc., 1987.

[10] G. Nelson and D. C. Oppen, "Simplification by Cooperating Decision Procedures," *ACM Trans. Programming Languages and Systems*, Vol. 1, pp. 245–257, October 1979.

[11] S. D. Crocker, *State Deltas: A Formalism for Representing Segments of Computation.* PhD thesis, University of California, Los Angeles, 1977.

[12] M. J. C. Gordon, *The Denotational Description of Programming Languages: An Introduction*, (New York: Springer-Verlag, 1979).

[13] D. F. Martin and J. V. Cook, "Adding Ada Program Verification Capability to the State Delta Verification System (SDVS)," in *Proceedings of the 11th National Computer Security Conference*, National Bureau of Standards/National Computer Security Center, October 1988.

[14] J. V. Cook, "Example Proofs of Stage 3 Ada Programs in the State Delta Verification System (SDVS)," Technical Report ATR-91(6778)-3, The Aerospace Corporation, September 1991.

[15] I. V. Filippenko, "A Formal Description of the Incremental Translation of Stage 2 VHDL into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-92(2778)-4, The Aerospace Corporation, September 1992.

[16] I. V. Filippenko, "Some Examples of Verifying Stage 3 VHDL Hardware Descriptions Using SDVS," Technical Report ATR-93(3778)-1, The Aerospace Corporation, September 1993.

[17] T. A. Aiken and D. F. Martin, "A Revised Formal Description of the Incremental Translation of ISPS into State Deltas in the State Delta Verification System (SDVS)," Technical Report ATR-90(5778)-1, The Aerospace Corporation, September 1990.

[18] J. V. Cook, "The Language for DENOTE (Denotational Semantics Translation Environment)," Technical Report TR-0090(5920-07)-2, The Aerospace Corporation, September 1990.

[19] L. Marcus and B. H. Levy, "Specifying and Proving Core VHDL Descriptions in the State Delta Verification System (SDVS)," Technical Report ATR-89(4778)-5, The Aerospace Corporation, September 1989.

[20] D. F. Martin, "A Preliminary Formal Description of the Incremental Translation of ISPS into State Deltas in the State Delta Verification System," Technical Report ATR-86A(2778)-7, The Aerospace Corporation, September 1987.

[21] J. V. Cook, "Test Suite for Static Semantic Errors in ISPS Descriptions," Technical Report ATR-89(4778)-3, The Aerospace Corporation, September 1989.

[22] B. H. Levy, "Inadequacies of ISPS as a Specification Language for Microcode Verification," Technical Report ATR-86A(2778)-1, The Aerospace Corporation, September 1987.